4-29-2010

Modeling Bound States in the Continuum for Multiple Electrons in a Quantum Dot Pair

Thomas Ian Tuegel
Butler University

Modeling Bound States in the Continuum for Multiple Electrons in a Quantum Dot Pair

A Thesis

Presented to the Department of Physics and Astronomy

College of Liberal Arts and Sciences

and

The Honors Program

of

Butler University

In Partial Fulfillment

of the Requirements for Graduation Honors

Thomas Ian Tuegel

April 29, 2010

Abstract

The bound eigenstates of an electron inside a pair of quantum dots embedded on an infinite quantum wire are examined using the method of particular solutions. The eigenstates of a two-electron system in the same structure are examined perturbatively using wavefunction expansions for the one-electron eigenstates. The stability of the two-electron eigenstates is evaluated and compared with other theoretical results. It is found that stable bound states in the continuum exist for two electrons confined to this geometry.

Contents

1 Introduction

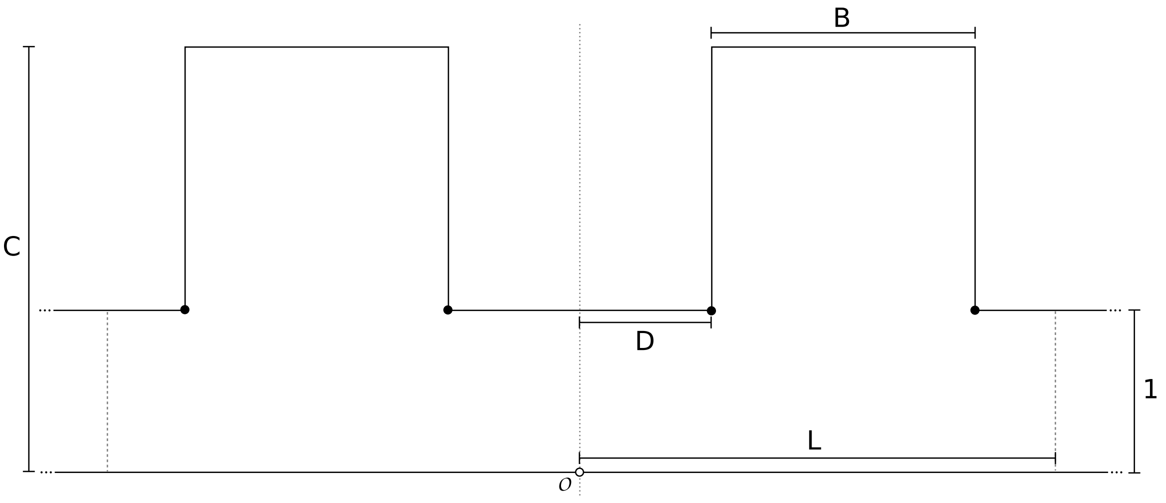
Quantum mechanics postulates that particles, such as electrons, exist not only as particles, but as waves. These waves represent the probability of the particle's existence at a particular point. A quantum wire can act as a guide for the electron's probability wave, constraining it to propagate in one dimension only. The introduction of a cavity–a region where it is allowed to propagate in two dimensions–to the wire causes the wave to build up, corresponding to a high probability of finding the electron inside the cavity. The trapped electron's wavefunction eventually decays, slowly increasing the probability that the electron will escape.

The strange picture of the electron as a wave becomes even more bizarre when a second cavity is introduced further down the wire. As the probability wave slowly dissipates from the first cavity through the wire, it encounters the second cavity, where is begins again to build up and gradually decay. If the separation between the cavities is a particular value, the decaying waves from both cavities interfere constructively between the cavities and destructively elsewhere. This corresponds to a very high probability of finding the electron trapped somewhere between the cavities and a very low probability of finding it anywhere else. These states are called bound states in the continuum because the electrons are trapped, even though they have energy states in the band of states which would normally allow them to escape along the wire.

Interest in two-dimensional quantum mechanics has peaked recently due in no small part to the development within the last two decades of fabrication techniques for semiconductor quantum wires, for example by deposition [10] or lithography [7]. It has also been shown that some quantum wire geometries have analogous microwave waveguide configurations [4]. These waveguide geometries are known to exhibit unintuitive behaviors which have not been investigated until relatively recently.

Previous work in this area by Ordonez et al. for a similar geometry suggests the

Figure 1: The quantum dot pair with relevant dimensions and coordinate origin labeled.

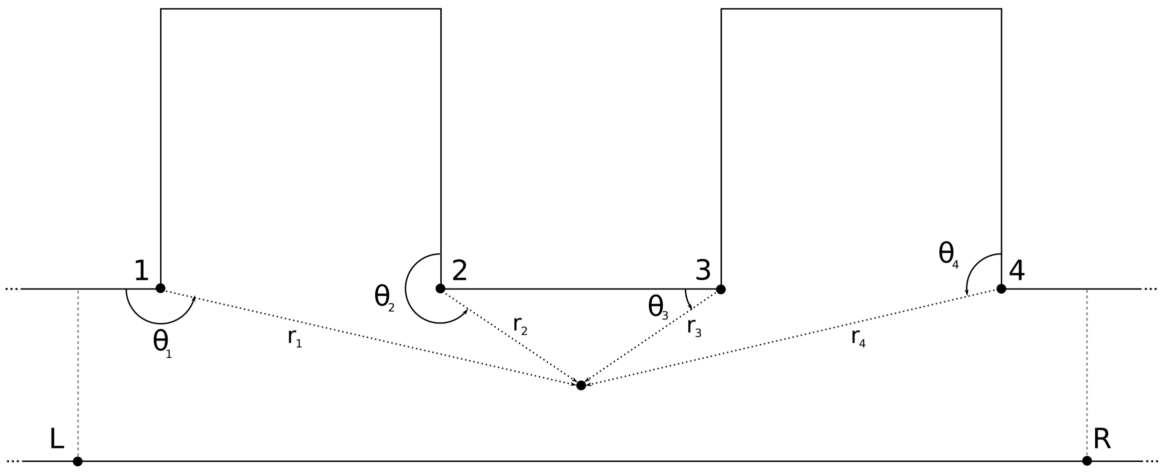


existence of bound states in the continuum for a single electron in this structure [8]. The work of Sadreev and Babushkina suggests that the existence of such bound states guarantee the existence of similar bound states for two electrons [9]; they consider the effect of the Coulomb interaction on the stability of two electron bound states in the continuum in general quantum dot structures using the Anderson model. The aim of this work, in part, is to confirm this finding for two electrons in this particular geometry.

2 Theory

Geometry The dimensions of the quantum dot structure are shown in Figure 1; note that the dimensions are normalized to the width of the wire. As shown in Figure 2, the wavefunction is modeled by one expansion in each lead and by the sum of four expansions in the center; the details of the expansions follow the discussion of the Helmholtz equation. The conditions that the wavefunction be zero on the boundary and continuously differentiable on an arbitrary boundary between expansion regions (indicated by the grey, dashed line in each lead) are imposed.

Figure 2: The quantum dot pair with coordinate origins and angular measurement references indicated.



The corner expansions were chosen to coincide with those corners having interior angle $\theta \neq \frac{\pi}{\beta}$ for some integer β, which are known to cause singularities with other expansion schemes. The reference angles for measuring the azimuthal coordinate are indicated in Figure 2. The branch cuts,

$$\theta_1 \in [0, 2\pi)$$
$$\theta_2 \in [0, 2\pi)$$
$$\theta_3 \in \Big(-\frac{\pi}{2}, \frac{3\pi}{2}\Big]$$
$$\theta_4 \in [0, 2\pi)$$

are chosen so that the introduced discontinuities fall outside the area of interest.

Helmholtz Equation The length units in the wire are normalized for convenience so the wire is of unit width. The time-independent free-particle Schrödinger equation becomes

$$-\frac{\hbar^2}{2m^*A^2}\nabla^2\Phi = E''\Phi \tag{1}$$

where A is the width of the wire in the original units, m^* is the effective mass of the electron, Φ is the single-electron wavefunction, and E'' is the eigenvalue. The form of the Helmholtz equation,

$$\left(\nabla^2 + k^2\right)\Phi = 0 \tag{2}$$

is recovered, where $k^2 = 2m^* A^2 E \hbar^{-2}$.

Lead Expansions Applying separation of variables,

$$\Phi\left(x, y\right) = \Phi_x\left(x\right)\Phi_y\left(y\right) \tag{3}$$

to the partial differential equation (2) renders the ordinary differential equations

$$\frac{1}{\Phi_y}\frac{\partial^2 \Phi_y}{\partial y^2} = -\omega^2 \tag{4}$$

$$\frac{1}{\Phi_x}\frac{\partial^2 \Phi_x}{\partial x^2} = \omega^2 - k^2 \tag{5}$$

where ω is the constant of separation. These equations permit obvious solutions: equation (4) is solved by

$$\Phi_y = \sin\left(\omega y\right) \tag{6}$$

where the choice of $\omega = j\pi$ satisfies the boundary conditions in the wire leads, and equation (5) has the solution

$$\Phi_x = e^{-\alpha_j x} \tag{7}$$

where $\alpha_j^2 = j^2\pi^2 - k^2$. Therefore the solution to the Helmholtz equation, away from the embedded dots, is modeled in the right lead by

$$\Phi_R = \sum_j a_j \sin\left(j\pi y_R\right) e^{-\alpha_j x_R} \tag{8}$$

and in the left by

$$\Phi_L = \sum_j b_j \sin(j\pi y_L) e^{-\alpha_j x_L}. \tag{9}$$

Corner Expansions To avoid singularity, the wavefunction in the central region is expanded in polar coordinates around each singular corner. Applying separation of variables,

$$\Phi = \Phi_r(r)\,\Phi_\theta(\theta) \tag{10}$$

and the substitution $u = kr$ to the partial differential equation (2) renders the ordinary differential equations

$$\frac{u}{\Phi_r}\frac{\partial}{\partial u}\left(u\frac{\partial \Phi_r}{\partial u}\right) = \omega^2 - u^2 \tag{11}$$

$$\frac{1}{\Phi_\theta}\frac{\partial^2 \Phi_\theta}{\partial \theta^2} = -\omega^2. \tag{12}$$

This yields the solution

$$\Phi_\theta = \sin(j\beta\theta) \tag{13}$$

where $\beta = \frac{2}{3}$ so that $\omega = j\beta$ and Φ_θ satisfies the condition that $\Phi = 0$ at the boundaries adjacent to the chosen corner. We also recognize that Equation (11) is solved by the ordinary Bessel function of the first kind, so that

$$\Phi_r = J_{j\beta}(kr). \tag{14}$$

Therefore, the solution around corner i is given by

$$\Phi_i = \sum_j c_{i,j} J_{j\beta}(kr_i)\sin(j\beta\theta_i). \tag{15}$$

Boundary Conditions The boundary conditions, in detail, are:

$$\sum_{i=1}^{4} \Phi_i (x, 0) = 0 \qquad \forall \pm x \in [0, L] \tag{16}$$

$$\sum_{i=1}^{4} \Phi_i (x, 1) = 0 \qquad \forall \pm x \in [0, D] \cup [D + B, L] \tag{17}$$

$$\sum_{i=1}^{4} \Phi_i (x, C) = 0 \qquad \forall \pm x \in [D, D + B] \tag{18}$$

$$\sum_{i=1}^{4} \Phi_i (x, y) = 0 \qquad \forall \pm x \in \{D, D + B\}, y \in [1, C]. \tag{19}$$

A finite number of points along these boundaries are selected and encoded in matrix form so that

$$A_B (k) \begin{bmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \end{bmatrix} = \vec{0}, \tag{20}$$

where each row of $A_B (k)$ corresponds to a boundary point and each entry therein to a wavefunction expansion term; the vectors $\vec{a}$, $\vec{b}$, and $\vec{c}$ contain the expansion coefficients. The continuity conditions are:

$$\Phi_L (x, y) + \Phi_R (x, y) - \sum_{i=1}^{4} \Phi_i (x, y) = 0 \qquad \forall \pm x = L, y \in [0, 1] \tag{21}$$

$$\frac{\partial \Phi_L}{\partial x} (x, y) + \frac{\partial \Phi_R}{\partial x} (x, y) - \sum_{i=1}^{4} \frac{\partial \Phi_i}{\partial x} (x, y) = 0 \qquad \forall \pm x = L, y \in [0, 1], \tag{22}$$

which can be arranged in matrix form as with the boundary conditions so that

$$A_C (k) \begin{bmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \end{bmatrix} = \vec{0}. \tag{23}$$

Observe that the matrices $A_B(k)$ and $A_C(k)$ are functions of k because their elements are wavefunction expansion terms.

Method of Particular Solutions Typically, the method of particular solutions would employ the boundary and continuity condition matrices in Equations (20) and (23) directly to find wavefunction expansion coefficients satisfying the boundary conditions; with large matrices this has the limitation of frequently selecting the spurious solution $\Phi = 0$. To ensure that $\Phi \neq 0$ somewhere, Betcke and Trefethen [3] introduce the matrix A_I such that

$$A_I(k)\begin{bmatrix}\vec{a}\\ \vec{b}\\ \vec{c}\end{bmatrix} = \begin{bmatrix}\Phi(x_1, y_1)\\ \Phi(x_2, y_2)\\ \cdots\end{bmatrix} \tag{24}$$

for some points (x_i, y_i) in the interior of the dots. They then perform the QR factorization so that

$$A(k)\vec{C} = \begin{bmatrix}A_B(k)\\ A_C(k)\\ A_I(k)\end{bmatrix}\begin{bmatrix}\vec{a}\\ \vec{b}\\ \vec{c}\end{bmatrix} = Q(k)R(k)\vec{C} = Q(k)\vec{D}(k) = \begin{bmatrix}Q_B(k)\\ Q_C(k)\\ Q_I(k)\end{bmatrix}\vec{D}(k) = \begin{bmatrix}\vec{0}\\ \vec{0}\\ \vec{?}\end{bmatrix} \tag{25}$$

where $Q(k)$ is an orthogonal matrix and $R(k)$ is a right-triangular matrix. Note that the vector $\vec{D}(k)$ has yet to be determined; in fact, it is not yet known for which values of k a nonzero vector $\vec{D}(k)$ even exists.

Betcke and Trefethen proceed to show that all solutions to the boundary value problem

$$\begin{bmatrix}Q_B(k)\\ Q_C(k)\end{bmatrix}\vec{D}(k) = \vec{0} \tag{26}$$

automatically satisfy the condition that

$$Q_I(k)\,\vec{D}(k) \neq \vec{0} \tag{27}$$

or, that Φ is nonzero *somewhere* [3]. Consider a unit vector $\tilde{v}$ where

$$\left\| \begin{bmatrix} Q_B(k) \\ Q_C(k) \end{bmatrix} \tilde{v} \right\| = \sigma(k). \tag{28}$$

If $\tilde{u} = Q(k)\,\tilde{v}$, then by the orthogonality of the matrix $Q(k)$, $\left\|\tilde{u}\right\|^2 = 1$. However, it is also the case that

$$\left\|\tilde{u}\right\|^2 = \left\| \begin{bmatrix} Q_B(k) \\ Q_C(k) \\ Q_I(k) \end{bmatrix} \tilde{v} \right\|^2 = \sigma(k)^2 + \left\|Q_I(k)\,\tilde{v}\right\|^2 \tag{29}$$

so that

$$\sigma(k)^2 + \left\|Q_I(k)\,\tilde{v}\right\|^2 = 1. \tag{30}$$

Therefore, as $\sigma(k) \to 0$, $\left\|Q_I\tilde{v}\right\|^2 \to 1$ so that Φ is nonzero inside the quantum wire when $\tilde{v}$ is a solution to the boundary value problem in Equation 26. The vector $\tilde{v}$ is the singular vector of $\begin{bmatrix} Q_B(k) \\ Q_C(k) \end{bmatrix}$ corresponding to the smallest singular value $\sigma(k)$. The singular values and singular vectors of a matrix can be determined from the singular value decomposition.

This implementation uses the singular value decomposition routine `zgesvd` provided by LAPACK [2]. D. E. Amos has developed the `zbesj` routine for evaluating Bessel functions of complex argument and real, nonnegative order, so that the $A(k)$ and $Q(k)$ matrices can be constructed [1]. The value of $\sigma(k)$ is minimized using the `gsl_multimin_fminimizer_nmsimplex2` minimizer of the GNU Scientific Library [5]. The corresponding singular vector determines the wavefunction coefficients of the approximate eigenfunction [3].

2-electron Eigenstates The eigenstates of the 2-electron system are determined by the equation

$$\left[-\frac{\hbar^2}{2m^*}\left(\nabla_1^2+\nabla_2^2\right)+\frac{V}{|\vec{r}_1-\vec{r}_2|}\right]\Psi = E\Psi \tag{31}$$

where E is an eigenvalue with corresponding eigenfunction Ψ, ∇_i^2 is the Laplace operator on the coordinates $(x_i, y_i) = \vec{r}_i$, and m^* is the effective mass of the electron. This equation can be solved perturbatively using the solutions to the unperturbed equation

$$-\frac{\hbar^2}{2m^*}\left(\nabla_1^2+\nabla_2^2\right)\Psi' = E'\Psi' \tag{32}$$

where $\Psi' = \Phi_j\Phi_k$ and $E' = E_j'' + E_k''$ and $j \neq k$ if the electrons have the same spin. The time-independent free-particle Schrödinger equation,

$$-\frac{\hbar^2}{2m^*}\nabla^2\Phi = E''\Phi \tag{33}$$

determines Φ and E''. The first-order energy perturbation is given by

$$E^{(1)} = \left\langle\Psi'\middle|\frac{V}{|\vec{r}_1-\vec{r}_2|}\middle|\Psi'\right\rangle \tag{34}$$

so that the existence of perturbed two-electron eigenstates is confirmed if the energy perturbation is entirely real (as unperturbed two-electron eigenstates clearly exist). Regardless of the electrons' spins, $E^{(1)}$ is entirely real for any bound states in the continuum (where Ψ is real).

3 Results

Results obtained from the method of particular solutions indicate the existence of bound states in the continuum at dot separations of $D = 1.75$. Figure 3 is a singular value plot obtained from the method of particular solutions by varying k, showing the existence of eigenvalues above the continuum energy, $k = 3.2070$. Figure 7 is a plot of the eigenfunction corresponding to $k = 3.5690$ showing the localization of the electron inside the dots characteristic of a bound state. The bound states in the continuum shown in Figures 8, 9, 10, and 11 also exhibit localization. For comparison, Figures 4 and 5 show the two true bound states below the continuum energy. These results confirm the previous work by Ordonez, et al., which indicated the presence of bound states in the continuum at this dot separation [8].

4 Conclusion

The results of this analysis based on the method of particular solutions indicate the existence of two-electron bound eigenstates analogous to the one-electron bound states in the continuum, as well as confirming previous results regarding the same one-electron bound states. Furthermore, the two-electron states are stable up to first-order perturbation for combinations of one-electron bound states in the continuum, as predicted by Sadreev and Babushkina [9]. The stability of these states suggests the application of such a quantum wire structure as a quantum memory device. Finally, this method allows for the investigation of complex eigenstates–such as with quasi-bound states in the continuum–and therefore warrants further investigation.

Figure 3: The variance of singular value under variance of eigenvalue k^2 with a dot separation of $D = 1.75$, showing the existence of real eigenvalues. The eigenvalues, as determined by the minimization routine, are indicated.

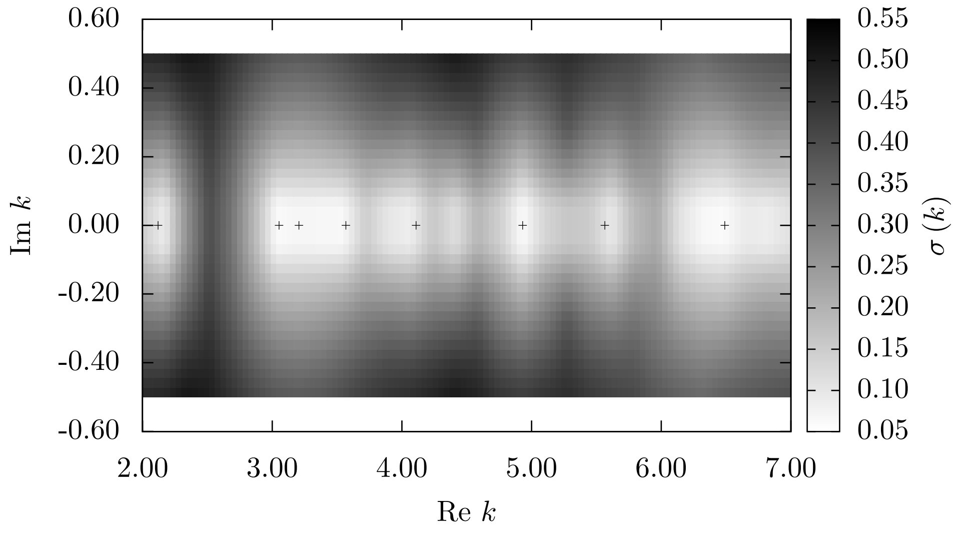


Figure 4: A plot of the probability associated with the first true bound eigenstate at $k = 2.1211$.

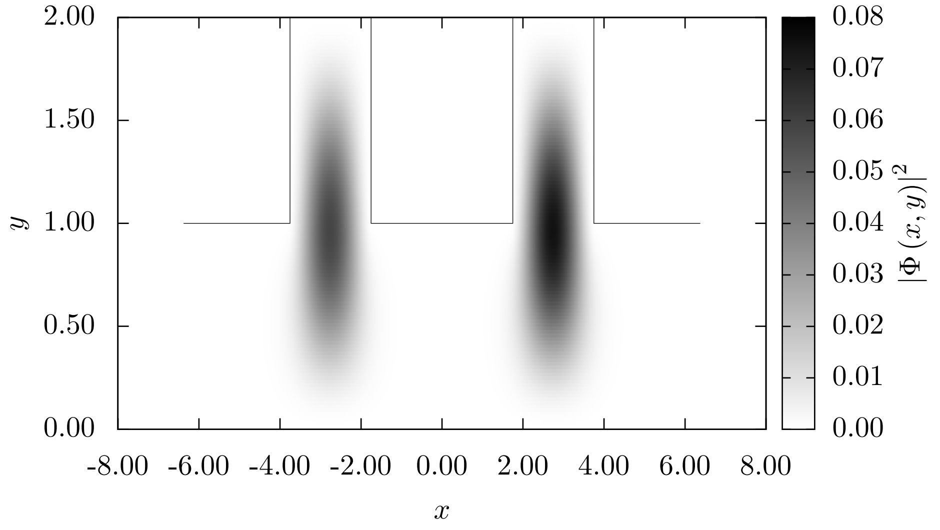


Figure 5: A plot of the probability associated with the second bound eigenstate at $k = 3.0539$.

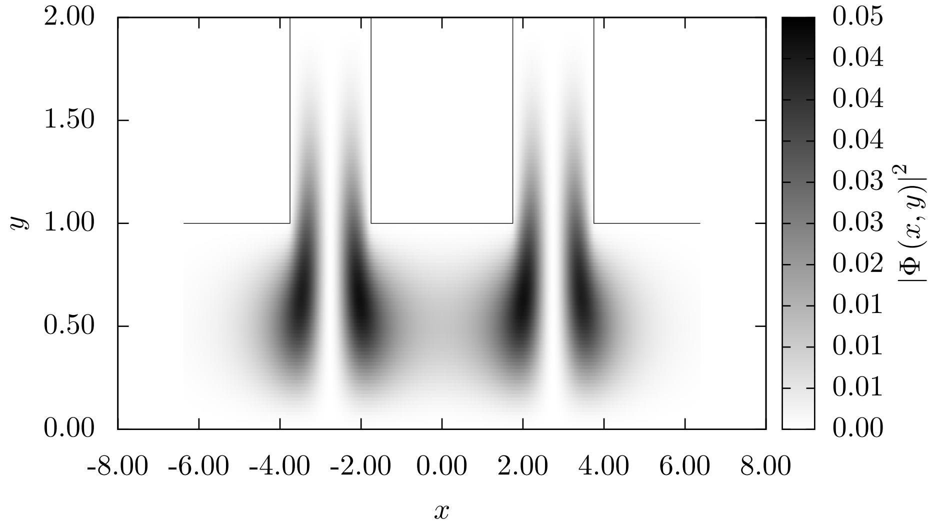


Figure 6: A plot of the probability associated with the continuum eigenstate at $k = 3.2070$.

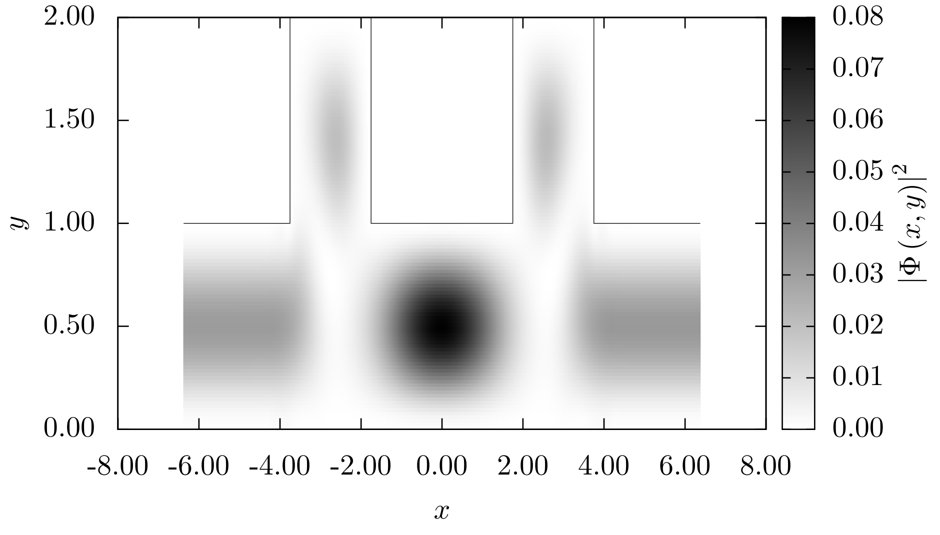


Figure 7: A plot of the probability associated with the first bound eigenstate in the continuum at $k = 3.5690$.

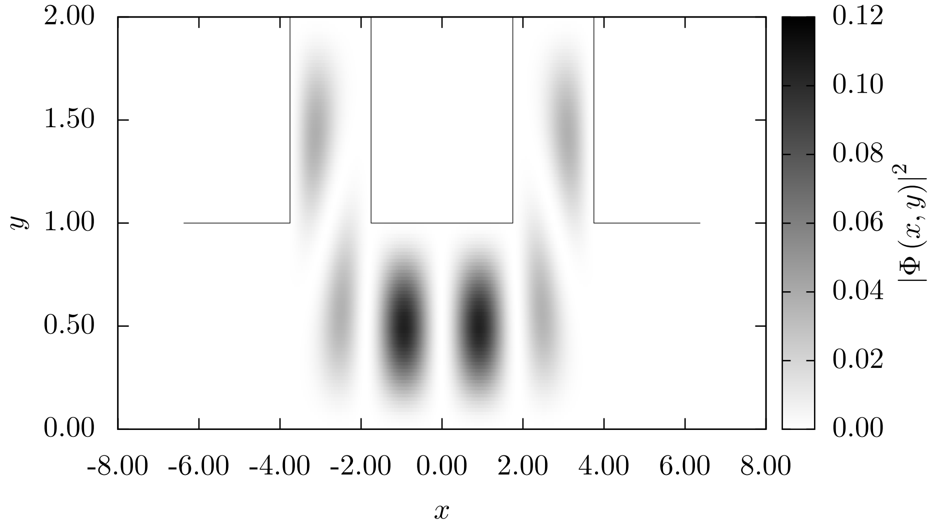


Figure 8: A plot of the probability associated with the second bound eigenstate in the continuum at $k = 4.1101$.

Figure 9: A plot of the probability associated with the third bound eigenstate in the continuum at $k = 4.9315$.

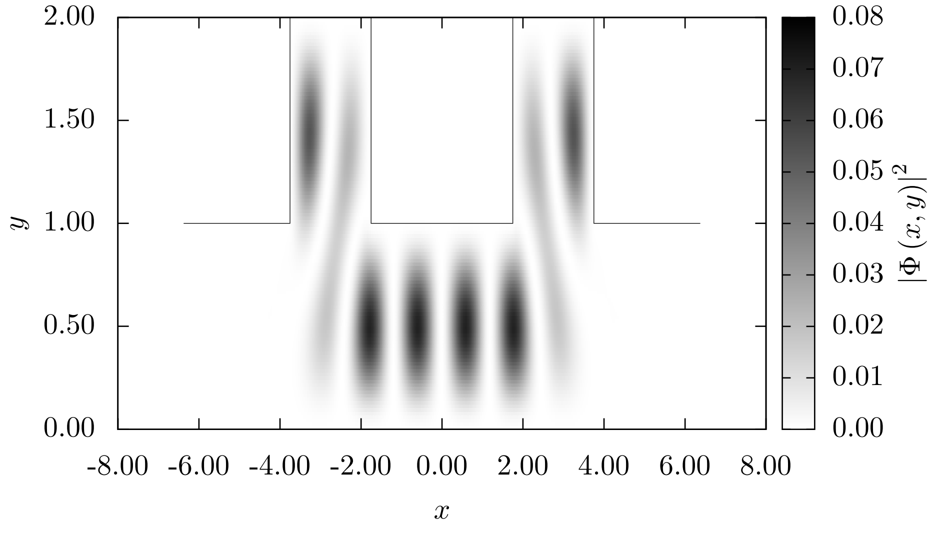


Figure 10: A plot of the probability associated with the fourth bound eigenstate in the continuum at $k = 5.5647$.

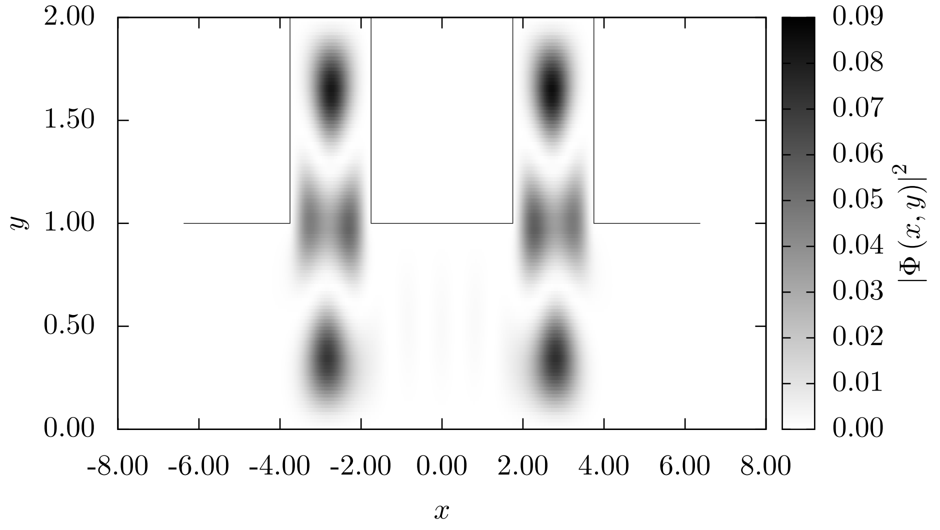
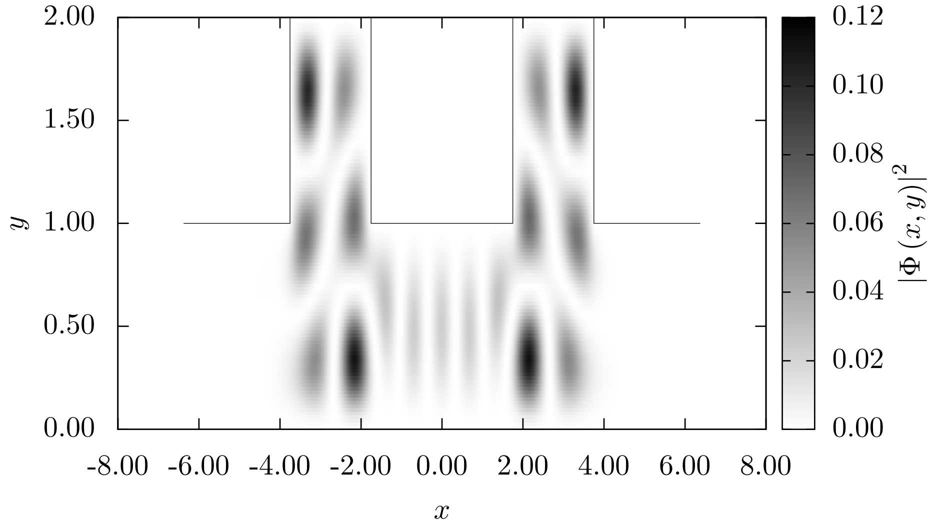


Figure 11: A plot of the probability associated with the fifth bound eigenstate in the continuum at $k = 6.4898$.



Acknowledgements

The author would like to acknowledge the invaluable assistance of his advisor, G. Ordonez of Butler University, as well as J. T. Londergan of Indiana University and D. Murdock of Tennessee Tech University, who advised a preliminary implementation of the method of particular solutions.

References

[1] D. E. Amos. A portable package for bessel functions of a complex argument and nonnegative order. *ACM Transactions on Mathematical Software*, 12:265–273, 1986.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] T. Betcke and L. N. Trefethen. Reviving the method of particular solutions. *SIAM Review*, 47(3):469–491, 2005.

[4] J. P. Carini, J. T. Londergan, K. Mullen, and D. P. Murdock. Bound states and resonances in waveguides and quantum wires. *Phys. Rev. B*, 46(23):15538–15541, 1992.

[5] M. Galassi et al. *GNU Scientific Library Reference Manual*. Third edition, 2009.

[6] S. P. Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[7] T. Nishida, M. Notomi, R. Iga, and T. Tamamura. Quantum wire fabrication by e-beam lithography using high-resolution and high-sensitivity e-beam resist zep-520. *Japanese Journal of Applied Physics*, 31:4508–4514, 1992.

[8] G. Ordonez, K. Na, and S. Kim. Bound states in the continuum in quantum-dot pairs. *Phys. Rev. A*, 73, 2006.

[9] A. F. Sadreev and T. V. Babushkina. Two-electron bound states in a continuum in quantum dots. *JETP Letters*, 88:312–317, 2008.

[10] X. Wang, M. Ogura, and H. Matsuhata. Fabrication of highly uniform algaas/-gaas quantum wire superlattices by flow rate modulation epitaxy on v-grooved substrates. *Journal of Crystal Growth*, 171:341–348, 1997.

A Code Listing

The implementation of this method was written in the language Haskell. Haskell [6] is a statically typed, purely functional programming language featuring polymorphic types, lazy evaluation, and automatic garbage collection. Haskell was chosen for this project because of the author's familiarity with the language, the ease with which Haskell can interface with existing codes in C and Fortran, the language's support for multiprocessor parallelism without programmer intervention, and the ease of rapid prototyping. This project took advantage of multiprocessor parallelism on Butler University's "BigDawg" computer cluster. A search of a typical spectrum space, such as in Figure 3, takes 10 to 12 hours.

A.1 Common

```
{-# LANGUAGE ForeignFunctionInterface,ScopedTypeVariables #-}
-- | This module handles evaluation of Bessel functions of the first kind with complex arguments, required to allow complex energy
      eigenvalues. The actual calculation is handled by the Fortran library to which this module interfaces through the C FFI.
module Numeric.AmosBesselJ ( besselJnus ) where

import Control.Monad.Stream
import Data.Complex
import Foreign
import Foreign.C.Types
import Foreign.Storable.Complex ()
import Prelude hiding (mapM)

foreign import ccall "zbesj_" zbesj :: Ptr CDouble -> Ptr CDouble -> Ptr CDouble -> Ptr CDouble -> Ptr CDouble -> Ptr CInt -> IO ()

-- | Bessel function of the first kind of real, positive order and complex argument.
besselJnus :: [Double] -> Complex Double -> [Complex Double]
besselJnus nus z = unsafePerformIO $
  alloca $ \pzr ->
    alloca $ \pzi ->
    alloca $ \pcyr ->
    alloca $ \pcyi ->
    alloca $ \pierr ->
    alloca $ \pnu -> do
      poke pzr $ floatCoerce $ realPart z
      poke pzi $ floatCoerce $ imagPart z
      poke pcyr 0.0
      poke pcyi 0.0
      poke pierr 0
      poke pnu 0.0
      let worker nu = do poke pnu $ floatCoerce nu
                         poke pierr 0
                         zbesj pzr pzi pnu pcyr pcyi pierr
                         throwErrors nu z $ peek pierr
                         liftM2 (:+) (fcm $ peek pcyr) (fcm $ peek pcyi)
          fcm = liftM floatCoerce
      mapM worker nus

floatCoerce :: (RealFloat a, RealFloat b) => a -> b
floatCoerce = uncurry encodeFloat . decodeFloat

throwErrors :: Double -> Complex Double -> IO CInt -> IO ()
throwErrors nu z i =
  do throwIf_ (== 1) (const inputError) i
     throwIf_ (== 2) (const "zbesj:␣Overflow.") i
     throwIf_ (== 3) (const "zbesj:␣Large␣argument␣degraded␣precision.") i
     throwIf_ (== 4) (const "zbesj:␣Large␣argument␣prevented␣calculation.") i
     throwIf_ (== 5) (const "zbesj:␣Termination␣condition␣not␣met.") i
  where inputError = "zbesj:␣Input␣error." ++ params
        params = "␣nu␣=␣" ++ show nu ++ ",␣z␣=␣" ++ show z


module Branch (Branch (..), mod') where

data Branch a = ClosedOpen a a
              | OpenClosed a a

size :: Num a => Branch a -> a
size (ClosedOpen l u) = u - l
size (OpenClosed l u) = u - l

inside :: Ord a => a -> Branch a -> Bool
inside x (ClosedOpen l u) = x >= l && x < u
inside x (OpenClosed l u) = x > l && x <= u

below :: Ord a => a -> Branch a -> Bool
below x (ClosedOpen l _) = x < l
below x (OpenClosed l _) = x <= l

mod' :: (Ord a, Floating a) => Branch a -> a -> a
mod' branch x
  | x `inside` branch = x
  | x `below` branch = mod' branch (x + size branch)
  | otherwise = mod' branch (x - size branch)


{-# OPTIONS_GHC -fno-warn-orphans #-}
-- This file contains orphan instances because Complex lacks an instance for Binary.
module Data.Binary.Complex where

import Data.Binary
import Data.Complex

instance (RealFloat a, Binary a) => Binary (Complex a) where
  put z = do put $ realPart z
             put $ imagPart z

  get = do r <- get
           i <- get
           return $ r :+ i


{-# LANGUAGE TypeFamilies, FlexibleContexts, UndecidableInstances #-}
module Geometry ( Geometry(..)
                ) where

import Data.List.Stream
import Data.Vector.Stream ()
```

```haskell
import Numeric.LinearAlgebra
import Prelude hiding (concat, concatMap, last, length, map, zip)

class Geometry g where
  nMatrixCols :: g -> Int
  nBoundaryRows :: g -> Int
  nInteriorRows :: g -> Int

  boundaryRows :: g -> Complex Double -> [[Complex Double]]

  interiorRows :: g -> Complex Double -> [[Complex Double]]

  generatePoints :: g -> g

  matrix :: g -> Complex Double -> Matrix (Complex Double)
  matrix g k = reshape n $ fromList
                $ concat $ concat [ boundaryRows g k, interiorRows g k ]
    where n = nMatrixCols g

  wavefunction :: g -> Complex Double -> Vector (Complex Double) -> [(Double, Double)] -> [Complex Double]

  singularValue :: g -> Complex Double -> Double
  singularValue g k = last $ toList s
    where (_, s, _) = svd $ takeRows (nBoundaryRows g) q
          (q, _) = qrEcon $ matrix g k

  singularVector :: g -> Complex Double -> Vector (Complex Double)
  singularVector g k = inv r <> subVector 0 (nMatrixCols g) (last $ toColumns ws)
    where (_, _, ws) = svd $ takeRows (nBoundaryRows g) q
          (q, r) = qrEcon $ matrix g k

qrEcon :: Field e => Matrix e -> (Matrix e, Matrix e)
qrEcon m = (q', r')
  where (q, r) = qr m
        q' = takeColumns (cols m) q
        r' = takeRows (cols m) r


{-# LANGUAGE TypeFamilies, FlexibleInstances, TemplateHaskell, EmptyDataDecls #-}
module OpenDots ( module Data.Default
                , module Geometry
                , OpenDots(..)
                , Expansion(..)
                , Segment(..)
                , expansions
                , segments
                , (!)
                , Coordinate(..)
                , Point
                ) where

import Branch
import Control.Arrow ((&&&), (^<<))
import Control.Monad.Stream (liftM)
import Data.Binary
import Data.Binary.Vector ()
import Data.Default
import Data.List.Stream
import Data.Pair
import qualified Data.Map as Map
import Data.Map ((!))
import Geometry
import Generics.Regular
import Generics.Regular.Functions.Binary hiding (Binary)
import Numeric.AmosBesselJ
import Numeric.LinearAlgebra
import Prelude hiding ((++), concat, concatMap, filter, length, map, replicate, take, zip, zipWith)
import System.IO.Unsafe (unsafePerformIO)
import System.Random.Mersenne

data Expansion
    = Corner1 | Corner2 | Corner3 | Corner4 | LeadL
    | LeadR deriving (Eq, Ord, Show, Enum)

instance Binary (Expansion) where
  put = put . fromEnum
  get = liftM toEnum get

expansions :: [Expansion]
expansions = enumFrom Corner1

data Segment
    = Bottom | LeadLTop | LeadLLeft | LeadRTop | LeadRRight | DotALeft
    | DotARight | DotATop | DotBLeft | DotBRight | DotBTop | Top
    deriving (Eq, Ord, Show, Enum)

instance Binary (Segment) where
  put = put . fromEnum
  get = liftM toEnum get

segments :: [Segment]
segments = enumFrom Bottom
```

```
data Coordinate = Global | Local Expansion deriving (Eq, Ord)

$(deriveAll ''Coordinate "PFCoordinate")
type instance PF Coordinate = PFCoordinate

instance Binary Coordinate where
    put = gput
    get = gget

type Point = Map.Map Coordinate (Pair Double)

data OpenDots = OpenDots { xBound :: Int
                         , yBound :: Int
                         , nContPts :: Int
                         , nIntPts :: Int
                         , nLeadTerms :: Int
                         , nCornerTerms :: Int
                         , dotWidth :: Double
                         , dotHeight :: Double
                         , dotSpacing :: Double
                         , leadLength :: Double
                         , boundPts :: Map.Map (Segment) [Point]
                         , intPts :: [Point] }

$(deriveAll ''OpenDots "PFOpenDots")
type instance PF OpenDots = PFOpenDots

instance Binary OpenDots where
  put = gput
  get = gget

instance Default OpenDots where
    def = OpenDots
        { xBound = 40
        , yBound = 20
        , nContPts = 40
        , nIntPts = 80
        , nLeadTerms = 12
        , nCornerTerms = 12
        , dotWidth = 2.0
        , dotHeight = 2.0
        , dotSpacing = 1.75
        , leadLength = 0.5
        , boundPts = Map.empty
        , intPts = []
        }

origin :: OpenDots -> Expansion -> Pair Double
origin od Corner1 = (-1.0 * (dotSpacing od + dotWidth od), 1.0)
origin od Corner2 = (-1.0 * dotSpacing od, 1.0)
origin od Corner3 = (dotSpacing od, 1.0)
origin od Corner4 = (dotSpacing od + dotWidth od, 1.0)
origin od LeadL = (-1.0 * (dotSpacing od + dotWidth od + leadLength od), 0.0)
origin od LeadR = (dotSpacing od + dotWidth od + leadLength od, 0.0)

toLocal :: OpenDots -> Pair Double -> Expansion -> Pair Double
toLocal od pt e | e == LeadL = (-x, y)
                | e == LeadR = (x, y)
                | otherwise = (r, h)
    where
        (x, y) = pt `psub` origin od e
        h = mod' (branch e) $ atan2 y x - offset e
        r = sqrt (x ** 2.0 + y ** 2.0)

fromGlobal :: OpenDots -> Pair Double -> Point
fromGlobal g point =
    Map.fromList $ concat
        [ (Global, point) : []
        , zip (map Local expansions) (map (toLocal g point) expansions)
        ]

instance Geometry OpenDots where
  nMatrixCols od = 2 * nLeadTerms od + 4 * nCornerTerms od
  nBoundaryRows od = nBoundPts + 2 * nContPts od
    where nBoundPts = length $ concat $ Map.elems $ boundPts od
  nInteriorRows = nIntPts

  boundaryRows od k = concatMap (boundaryRows' od k) segments

  interiorRows od k = map row $ intPts od
    where row v = concat [ concatMap (\n -> terms od n k v) $ enumFromTo Corner1 Corner4, leadBuffer od, leadBuffer od ]

  generatePoints od = od { boundPts = uniformBoundaryPoints od, intPts = randomInteriorPoints od }

  wavefunction od k coef rs = toList $ mat <> coef
    where mat = fromLists $ map wf rs
          wf r | interior od r = cornerRow od k $ fromGlobal od r
               | inLeadL od r = leadRow od LeadL k $ fromGlobal od r
               | inLeadR od r = leadRow od LeadR k $ fromGlobal od r
               | otherwise = zeroRow
          zeroRow = concat [ cornersBuffer od, leadBuffer od, leadBuffer od ]
```

```
boundaryRows' :: OpenDots -> Complex Double -> Segment -> [[Complex Double]]
boundaryRows' od k s | s == LeadLLeft = concat [ map (leadBoundRow LeadL) pts, map (leadDerivRow LeadL) pts ]
                     | s == LeadRRight = concat [ map (leadBoundRow LeadR) pts, map (leadDerivRow LeadR) pts ]
                     | otherwise = map (cornerRow od k) pts
    where leadBoundRow ex = uncurry sub ^<< cornerRow od k &&& leadRow od ex k
          leadDerivRow = ddX od . leadBoundRow
          pts = boundPts od ! s

branch :: Expansion -> Branch Double
branch Corner1 = ClosedOpen 0 (2.0 * pi)
branch Corner2 = ClosedOpen 0 (2.0 * pi)
branch Corner3 = OpenClosed (- pi / 2.0) (3.0 * pi / 2.0)
branch Corner4 = ClosedOpen 0 (2.0 * pi)
branch _ = error "OpenDots.branch:␣Argument␣must␣be␣a␣corner."

offset :: Expansion -> Double
offset Corner1 = pi
offset Corner2 = pi / 2.0
offset Corner3 = pi
offset Corner4 = pi / 2.0
offset _ = error "OpenDots.offset:␣Argument␣must␣be␣a␣corner."

cornerRow :: OpenDots -> Complex Double -> Point -> [Complex Double]
cornerRow od k r = concat [ concatMap (\each -> terms od each k r) $ enumFromTo Corner1 Corner4
                          , leadBuffer od, leadBuffer od ]

leadRow :: OpenDots -> Expansion -> Complex Double -> Point -> [Complex Double]
leadRow od ex k r = concat [ cornersBuffer od
                           , if ex == LeadL then ts else buf
                           , if ex == LeadR then ts else buf ]
  where ts = terms od ex k r
        buf = leadBuffer od

leadBuffer :: OpenDots -> [Complex Double]
leadBuffer od = replicate (nLeadTerms od) (0.0 :+ 0.0)

cornersBuffer :: OpenDots -> [Complex Double]
cornersBuffer od = replicate (4 * nCornerTerms od) (0.0 :+ 0.0)

interior :: OpenDots -> Pair Double -> Bool
interior od (x, y) = ((abs x < dotSpacing od + dotWidth od + leadLength od) && (y < 1.0)) ||
                  ((y < dotHeight od) && (abs x < dotSpacing od + dotWidth od) && (abs x > dotSpacing od))

inLeadL :: OpenDots -> Pair Double -> Bool
inLeadL od (x, y) = (x <= -(dotSpacing od) - dotWidth od - leadLength od) && (y <= 1.0)

inLeadR :: OpenDots -> Pair Double -> Bool
inLeadR od (x, y) = (x >= dotSpacing od + dotWidth od + leadLength od) && (y <= 1.0)

terms :: OpenDots -> Expansion -> Complex Double -> Point -> [Complex Double]
terms od e k pt = if leads then leadTerms ms e k pt else cornerTerms ms e k pt
  where m = if leads then nLeadTerms od else nCornerTerms od
        ms = map fromIntegral $ enumFromTo 1 m
        leads = e == LeadL || e == LeadR

cornerTerms :: [Double] -> Expansion -> Complex Double -> Point -> [Complex Double]
cornerTerms ms expansion k v | h == 0.0 || h == 3.0 * pi / 2.0 = map (:+ 0.0) $ scale 0.0 ms
                             | otherwise = mul bessels besselSines
  where (r, h) = v ! Local expansion
        nus = scale (2.0 / 3.0) ms
        bessels = besselJnus nus (k * (r :+ 0.0))
        besselSines = map zin $ scale (2.0 / 3.0 * h) ms
        zin = (:+ 0.0) . sin

leadTerms :: [Double] -> Expansion -> Complex Double -> Point -> [Complex Double]
leadTerms ms expansion k v = mul exps expSines
  where (x, y) = v ! Local expansion
        exps = map exp $ scale ((-x) :+ 0.0) $ map alpha ms
        alpha j = sqrt $ ((j * pi * j * pi) :+ 0.0) - (k * k)
        expSines = map zin $ scale (pi * y) ms
        zin = (:+ 0.0) . sin

ddX :: OpenDots -> (Point -> [Complex Double]) -> Point -> [Complex Double]
ddX od f v = scaleRecip ((2.0 * delta) :+ 0.0) $ sub (f d2) (f d1)
  where d2 = fromGlobal od $ (x + delta, y)
        d1 = fromGlobal od $ (x - delta, y)
        (x, y) = v ! Global
        delta = 1.0E-10

uniformBoundXs :: OpenDots -> Int -> [Double]
uniformBoundXs od n = map ((+ (-(leadLength od) - dotSpacing od - dotWidth od)) . (* int)) ns
  where ns = map fromIntegral $ enumFromTo 1 n
        int = 2.0 * (leadLength od + dotSpacing od + dotWidth od) / (fromIntegral n + 1)

uniformBoundYs :: OpenDots -> Int -> [Double]
uniformBoundYs od n = map (* int) ns
  where ns = map fromIntegral $ enumFromTo 1 n
        int = dotHeight od / (fromIntegral n + 1)

continuityYs :: Int -> [Double]
continuityYs n = map (* int) ns
```

```haskell
  where ns = map fromIntegral $ enumFromTo 1 n
        int = 1.0 / (fromIntegral n + 1)

uniformBoundaryPoints :: OpenDots -> Map.Map Segment [Point]
uniformBoundaryPoints od = Map.fromList $ zip segments $ map (map (fromGlobal od) . points od) segments

points :: OpenDots -> Segment -> [Pair Double]
points od c =
  let pts | c == Bottom = map (flip pair 0.0) boundXs
          | c == LeadLTop = map (flip pair 1.0) boundXs
          | c == LeadLLeft = map (padd (origin od LeadL) . pair 0.0) $ continuityYs $ nContPts od
          | c == Top = map (flip pair 1.0) boundXs
          | c == LeadRTop = map (flip pair 1.0) boundXs
          | c == LeadRRight = map (padd (origin od LeadR) . pair 0.0) $ continuityYs $ nContPts od
          | c == DotATop = map (flip pair $ dotHeight od) boundXs
          | c == DotBTop = map (flip pair $ dotHeight od) boundXs
          | c == DotARight = map (pair $ -(dotSpacing od)) boundYs
          | c == DotALeft = map (flip psub (dotWidth od, 0)) $ points od DotARight
          | c == DotBLeft = map (padd (2.0 * dotSpacing od, 0)) $ points od DotARight
          | c == DotBRight = map (padd (dotWidth od, 0)) $ points od DotBLeft
          | otherwise = undefined
  in filter (onSegment od c) pts
  where boundXs = uniformBoundXs od $ xBound od
        boundYs = uniformBoundYs od $ yBound od

onSegment :: OpenDots -> Segment -> Pair Double -> Bool
onSegment od s pt
  | s == Bottom = True
  | s == LeadLTop = -x >= dotSpacing od + dotWidth od
  | s == LeadLLeft = y < 1.0
  | s == Top = abs x <= dotSpacing od
  | s == LeadRTop = x >= dotSpacing od + dotWidth od
  | s == LeadRRight = y < 1.0
  | s == DotATop = -x <= dotSpacing od + dotWidth od && -x >= dotSpacing od
  | s == DotBTop = x <= dotSpacing od + dotWidth od && x >= dotSpacing od
  | s == DotARight = y > 1.0 && y < dotHeight od
  | otherwise = True
  where (x, y) = pt

randomInteriorPoints :: OpenDots -> [Point]
randomInteriorPoints od = unsafePerformIO $ do
  xs <- liftM (map ((+ offsetX) . (* scaleX))) $ getStdGen >>= randoms
  ys <- liftM (map (* scaleY)) $ getStdGen >>= randoms
  let pts = zip xs ys
  return $ map (fromGlobal od) $ take n $ filter (interior od) pts
  where scaleX = 2.0 * (dotSpacing od + dotWidth od + leadLength od)
        offsetX = -(dotSpacing od) - dotWidth od - leadLength od
        scaleY = dotHeight od
        n = nIntPts od


-- | This module is used in finding eigenvalues of the Hamiltonian for a 'Geometry'.  First, a 'spectrum' must be generated, giving
       minimum singular values at a sampling of trial eigenvalues.  The function 'eigenvalues' uses this information to find regions
       probably containing actual eigenvalues and minimizes the singular value function of the geometry in those regions to identify them.
module Spectrum ( Spectrum(..)
                , spectrum
                , plottableSpectrum
                , neighborhood
                , interval
                , onEdge
                , targets
                , eigenvalues
                , module Data.Default ) where

import Data.Binary
import Data.Binary.Complex ()
import Data.Default
import Geometry
import Numeric.GSL hiding (si)
import Numeric.LinearAlgebra hiding (eigenvalues, singularValues)

data Spectrum = Spectrum { minRe :: Double
                         , maxRe :: Double
                         , minIm :: Double
                         , maxIm :: Double
                         , resRe :: Int
                         , resIm :: Int
                         , spec :: [(Complex Double, Double)]
                         }

instance Binary Spectrum where
  put s = do put $ minRe s
             put $ maxRe s
             put $ minIm s
             put $ maxIm s
             put $ resRe s
             put $ resIm s
             put $ spec s

  get = do nr <- get
           xr <- get
           ni <- get
           xi <- get
           sr <- get
           si <- get
           s <- get
           return Spectrum { minRe = nr
                            , maxRe = xr
                            , minIm = ni
                            , maxIm = xi
                            , resRe = sr
                            , resIm = si
                            , spec = s
                            }

instance Default Spectrum where
  def = Spectrum  { minRe = 0.0
                  , maxRe = 9.0
                  , minIm = -0.5
                  , maxIm = 0.5
                  , resRe = 30
                  , resIm = 10
                  , spec = []
                  }

neighborhood :: Spectrum -> Complex Double -> [(Complex Double, Double)]
neighborhood s z = filter (\(w, _) -> magnitude (z - w) < interval s) $ spec s

interval :: Spectrum -> Double
interval s = 2.0 * magnitude (r :+ im)
  where r = (maxRe s - minRe s) / fromIntegral (resRe s)
        im = (maxIm s - minIm s) / fromIntegral (resIm s)

onEdge :: Spectrum -> Complex Double -> Bool
onEdge s z = z `elem` edges
  where realEdge z' = let r = realPart z' in r == maxRe s || r == minRe s
        imagEdge z' = let im = imagPart z' in im == maxIm s || im == minIm s
        edges = filter (\z' -> realEdge z' || imagEdge z') $ map fst $ spec s

spectrum :: Geometry g => g
         -> (Int, Int)
         -> (Double, Double)
         -> (Double, Double)
         -> [(Complex Double, Double)]
spectrum geom (xSamples, ySamples) (xLower, xUpper) (yLower, yUpper) =
  zip trialValues singularValues
  where trialValues = [ x :+ y | x <- realSamples, y <- imagSamples ]
        realSamples = map ((+ xLower) . (* xInterval) . fromIntegral)
                      $ enumFromTo 0 (xSamples - 1)
        imagSamples = map ((+ yLower) . (* yInterval) . fromIntegral)
                      $ enumFromTo 0 (ySamples - 1)
        xInterval = (xUpper - xLower) / fromIntegral (xSamples - 1)
        yInterval = (yUpper - yLower) / fromIntegral (ySamples - 1)
        singularValues = map (singularValue geom) trialValues

plottableSpectrum :: [(Complex Double, Double)] -> [Vector Double]
plottableSpectrum = map (\(z, s) -> fromList [ realPart z, imagPart z, s ])

targets :: Spectrum
        -> [Complex Double]
targets s = filter (not . onEdge s) $ map fst $ filter isMinimum $ filter ((<= 0.6) . snd) $ spec s
  where isMinimum (z, sing) = sing <= minimum (map snd (neighborhood s z))

eigenvalues :: (Complex Double -> Double)
            -> Spectrum
            -> [Complex Double]
eigenvalues singValue s = map eigenvalue $ targets s
  where eigenvalue z = toC $ fst $ minimize NMSimplex2 precision iterations [interval s, interval s] f [realPart z, imagPart z]
        f = singValue . toC
        precision = 1E-10
        toC xs = head xs :+ head (tail xs)
        iterations = 1000


{-# OPTIONS_GHC -fno-warn-orphans #-}
-- This file contains orphan instances which are missing from hmatrix for vectors as lists.
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, FlexibleContexts, UndecidableInstances #-}
module Data.Vector.Stream where

import Control.Arrow ((&&&))
import Data.List.Stream
import Numeric.LinearAlgebra
import Prelude hiding (map, unzip, zipWith)

instance Container [] Double where
  toComplex (re, im) = zipWith (:+) re im
  fromComplex = unzip . map (realPart &&& imagPart)
  comp = map (:+ 0.0)
  conj = map conjugate
  real = id
  complex = comp

instance Container [] (Complex Double) where
  toComplex (re, im) = zipWith (:+) re im
  fromComplex = unzip . map (realPart &&& imagPart)
  comp = map (:+ 0.0)
  conj = map conjugate
  real = comp
  complex = id

instance (Container [] e) => Linear [] e where
  scalar x = [x]
  scale x = map (* x)
  addConstant x = map (+ x)
  add = zipWith (+)
  sub = zipWith (-)
  mul = zipWith (*)
  divide = zipWith (/)
  scaleRecip x = map (/ x)
  equal = (==)
```

```
{-# OPTIONS_GHC -fno-warn-orphans #-}
-- This file contains orphan instances because hmatrix doesn't provide a Binary instance for Vector.
module Data.Binary.Vector where

import Control.Monad
import Data.Binary
import Numeric.LinearAlgebra

instance (Element e, Binary e) => Binary (Vector e) where
  put = put . toList

  get = liftM fromList get
```

A.2 Executables

```
{-# LANGUAGE ScopedTypeVariables #-}
module Main where

import Control.Parallel.Strategies
import Data.Binary
import Data.Binary.Complex ()
import Geometry
import OpenDots
import Spectrum
import System.Environment (getArgs)

main :: IO ()
main = do (file:_) <- getArgs
          findEigenvalues $ '-' : file

findEigenvalues :: String -> IO ()
findEigenvalues name = do (pts :: Spectrum) <- decodeFile $ "spectrum" ++ name ++ ".bin"
                          (od :: OpenDots) <- decodeFile $ "geometry" ++ name ++ ".bin"
                          let regs = targets pts
                              evs = flip using (parList rdeepseq) $ eigenvalues (singularValue od) pts
                              svs = map (singularValue od) evs
                          putStrLn "Targets:"
                          putStrLn $ unlines $ map show regs
                          putStrLn "(Eigenvalue,␣Singular␣Value)"
                          putStrLn $ unlines $ map show $ zip evs svs
                          encodeFile ("eigenvalues" ++ name ++ ".bin") evs
```

```
module Main where

import Data.Binary
import Data.Binary.Complex ()
import OpenDots
import System.Console.GetOpt
import System.Environment (getArgs)

options :: [OptDescr (OpenDots -> OpenDots)]
options =
  [ Option "t" ["dotsterms"] (ReqArg (\n o -> o { nCornerTerms = read n }) "INT")      "number␣of␣dot␣expansion␣terms"
  , Option "l" ["leadterms"] (ReqArg (\n o -> o { nLeadTerms = read n }) "INT")      "number␣of␣lead␣expansion␣terms"
  , Option "i" ["interior"]  (ReqArg (\n o -> o { nIntPts = read n }) "INT") "number␣of␣interior␣points"
  , Option "x" ["xBound"]    (ReqArg (\n o -> o { xBound = read n }) "INT")         "number␣of␣x␣boundary␣grid␣steps"
  , Option "y" ["yBound"]    (ReqArg (\n o -> o { yBound = read n }) "INT")          "number␣of␣y␣boundary␣grid␣steps"
  , Option "d" ["interdot"]  (ReqArg (\d o -> o { dotSpacing = read d }) "DOUBLE")    "edge-to-edge␣dot␣separation"
  , Option "w" ["width"]     (ReqArg (\d o -> o { dotWidth = read d }) "DOUBLE")     "dot␣width"
  , Option "h" ["height"]    (ReqArg (\d o -> o { dotHeight = read d }) "DOUBLE")    "dot␣height"
  , Option "c" ["cont"]      (ReqArg (\n o -> o { nContPts = read n}) "INT") "continuity␣points"
  ]

main :: IO ()
main = do argv <- getArgs
          case getOpt' Permute options argv of
            (o, [f], _, []) -> runOpenDots f $ foldl (flip id) def o
            (_, _, _, er) -> error $ concat er
```

```haskell
runOpenDots :: String -> OpenDots -> IO ()
runOpenDots file od = let geometryFile = "geometry-" ++ file
                      in encodeFile (geometryFile ++ ".bin") $ generatePoints od
```

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
module Main where

import Control.Parallel.Strategies
import Data.Binary
import Data.Binary.Complex ()
import OpenDots
import Spectrum
import System.Console.GetOpt
import System.Environment (getArgs)

options :: [OptDescr (Spectrum -> Spectrum)]
options =
  [ Option "x" ["minre"] (ReqArg (\d o -> o { minRe = read d }) "DOUBLE") "min.␣real␣component␣for␣k␣values"
  , Option "y" ["maxre"] (ReqArg (\d o -> o { maxRe = read d }) "DOUBLE") "max.␣real␣component␣for␣k␣values"
  , Option "u" ["minim"] (ReqArg (\d o -> o { minIm = read d }) "DOUBLE") "min.␣imaginary␣component␣for␣k␣values"
  , Option "v" ["maxim"] (ReqArg (\d o -> o { maxIm = read d }) "DOUBLE") "max.␣imaginary␣component␣for␣k␣values"
  , Option "r" ["resre"] (ReqArg (\n o -> o { resRe = read n }) "INT")    "k␣spectrum␣real␣resolution"
  , Option "s" ["resim"] (ReqArg (\n o -> o { resIm = read n }) "INT")    "k␣spectrum␣imaginary␣resolution"
  ]

main :: IO ()
main = do argv <- getArgs
          case getOpt' Permute options argv of
            (o, [f], _, []) -> runOpenDots f $ foldl (flip id) def o
            (_, _, _, er) -> error $ concat er

runOpenDots :: String -> Spectrum -> IO ()
runOpenDots file s = do let geometryFile = "geometry-" ++ file
                        (od :: OpenDots) <- decodeFile (geometryFile ++ ".bin")
                        let pts = flip using (parList rdeepseq)
                                      $ spectrum od (resRe s, resIm s) (minRe s, maxRe s) (minIm s, maxIm s)
                            spectrumFile = "spectrum-" ++ file
                        encodeFile (spectrumFile ++ ".bin") $ s { spec = pts }
```

```haskell
module Main where

import Control.Monad.Stream (liftM)
import Data.Binary
import Data.Binary.Complex ()
import Data.Complex
import Data.List.Stream
import Prelude hiding ((++), filter, map, unlines, unwords)
import Spectrum
import System.Environment (getArgs)
import System.IO

main :: IO ()
main = do
    (file:_) <- getArgs
    let name = "spectrum-" ++ file
    s <- liftM spec $ decodeFile $ name ++ ".bin"
    withFile (name ++ ".txt") WriteMode $ flip hPutStr $
         unlines $ map (unlines . map (\(x :+ y, z) -> unwords [show x, show y, show z])) $ -- Data file formatted for gnuplot: each
       line is a point with coordinates separated by spaces and each block is a scanline, with blocks separated by blank lines.
         map (\x -> filter ((== x) . realPart . fst) s) $   -- Scanlines as recognized by gnuplot: lines of constant 'x'.
         nub $ map (realPart . fst) s   -- Unique 'x' values.
```

```haskell
module Main where

import Data.Binary
import Data.Binary.Complex ()
import Data.List.Stream
import Geometry
import Numeric.LinearAlgebra
import OpenDots
import Prelude hiding ((++), filter, head, map, unlines, unwords, zipWith)
import System.Environment (getArgs)
import System.IO

main :: IO ()
main = do (file:_) <- getArgs
          let geomFile = "geometry-" ++ file
              evsFile = "eigenvalues-" ++ file
          od <- decodeFile $ geomFile ++ ".bin"
          evs <- decodeFile $ evsFile ++ ".bin"
          mapM_ (plotWavefunction file od) evs

plotWavefunction :: String -> OpenDots -> Complex Double -> IO ()
plotWavefunction file od k =
    let
        wf = wavefunction od k (singularVector od k) grid
        grid = [ (x, y) | x <- xs, y <- ys ]
        xMax = 1.5 * (dotSpacing od + dotWidth od + leadLength od)
        xs = toList $ linspace 200 (-xMax, xMax)
        ys = toList $ linspace 30 (0.0, dotHeight od)
        pts = zipWith (\(x, y) z -> [x, y, magnitude z ** 2.0]) grid wf
    in do
        withFile ("wavefunction-" ++ file ++ "-k_" ++ show k ++ ".txt") WriteMode $ flip hPutStr $
            unlines $ map (unlines . map (unwords . map show)) $ -- Data file formatted for gnuplot: each line is a point with
      coordinates separated by spaces and each block is a scanline, with blocks separated by blank lines.
            map (\x -> filter ((== x) . head) pts) $   -- Scanlines as recognized by gnuplot: lines of constant 'x'.
            nub $ map head pts   -- Unique 'x' values.
```

A.3 Utilities

```
#!/bin/bash
cd zbesj
gfortran -O3 -c *.f
cd ..
export GHC="ghc␣-Wall␣-threaded␣-feager-blackholing␣--make␣-O2␣-fexcess-precision␣-fvia-C␣-optc-O3␣-lgfortran"
${GHC} GenerateGeometry.hs ./zbesj/*.o
${GHC} GenerateSpectrum.hs ./zbesj/*.o
${GHC} FindEigenvalues.hs ./zbesj/*.o
${GHC} PlotSpectrum.hs
${GHC} PlotBoundary.hs ./zbesj/*.o
${GHC} PlotWavefunctions.hs ./zbesj/*.o
${GHC} DumpMatrix.hs ./zbesj/*.o
${GHC} CheckComplete.hs ./zbesj/*.o
```