

PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Boundary-value and eigenvalue problems

- We've completed solutions to solve “unbounded” ODE's
- Now we turn to boundary-value problems and eigenvalue problems
- These are closely related (of course, they solve the same mathematical constructs)
- Key complication : the boundary values must be met, so “marching” methods like RK4 are not always the most accurate

“1d PDEs”

Boundary-value and eigenvalue problems

- Consider a second-order ODE :

$$\frac{d^2 u}{dx^2} = f(x, u, u'), \quad u'(x) = du/dx$$

- We specify values on two boundaries (left and right)

$$x_{lb} \leq x \leq x_{rb}$$

- We can have :
 - Dirichlet : specify $u(x)$ on the boundaries
 - Neumann : specify $u'(x)$ on the boundaries
 - Periodic : specify $u(x_{lb}) = u(x_{rb})$, $u'(x_{lb}) = u'(x_{rb})$
 - Mixed are also possible

Boundary-value and eigenvalue problems

- Can also consider the eigenvalue problems :

$$\frac{d^2u}{dx^2} = f(x, u, u', \lambda) ,$$

- We've already encountered them earlier in the semester, too
- Will build upon the matrix methods we've already established!

Boundary-value and eigenvalue problems

- Example : time-independent Schroedinger equation :

$$\frac{d^2\psi}{dx^2} = -\frac{2m}{\hbar^2} [E - V(x)] \psi(x) ,$$

- Say $V(x)$ is a potential well :

$$\begin{aligned} V(x) &= 0 \text{ for } |x| < L \\ &= \infty \quad x \geq L \end{aligned}$$

- We have Dirichlet boundary conditions :

$$\psi(0) = 0, \psi(L) = 0$$

Boundary-value and eigenvalue problems

- Analytically, we have within the well :

$$\frac{d^2\psi}{dx^2} = -\frac{2m}{\hbar^2} E\psi(x)$$

- This is a free particle, so we guess

$$\psi(x) = Ae^{ikx} + Be^{-ikx}$$

- Applying the boundary conditions we get

$$\psi(0) = 0 = A + B$$

$$\psi(L) = 0 = 2A \sin kL = 0$$

$$k = \frac{n\pi}{L}$$

Boundary-value and eigenvalue problems

- Applying normalization :

$$\begin{aligned} 1 &= \int_0^L dx \ 4A^2 \sin^2\left(\frac{n\pi}{L}x\right) \\ &= 4A^2 \left(\frac{x}{2} - \frac{\sin\left(\frac{n\pi}{L}x\right)}{4n\pi/L} \right) \Big|_0^L \end{aligned}$$

–So we get : $\psi(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi}{L}x\right)$

–Energy eigenvalues are $E = \hbar\omega = \frac{\hbar^2 k^2}{2m}$

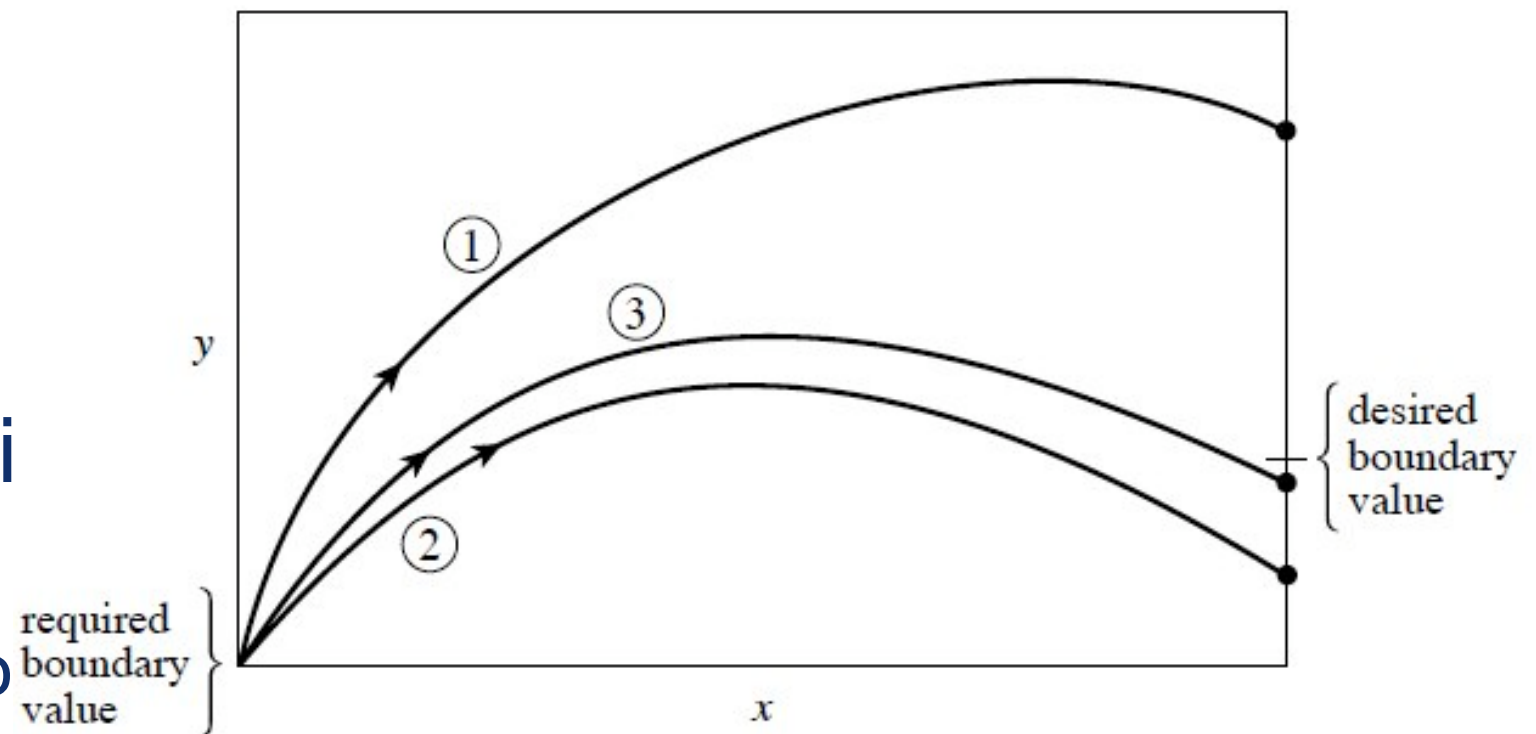
Boundary-value and eigenvalue problems

- General strategy is to either “shoot” or “relax” :
- “Shoot” : pick values via a guess, “shoot” the ODE to the other side, and correct iteratively
- “Relax” : pick values via a guess, check points on the interior, and relax until the correction is small
- When in doubt of which to use, from NR : Be a gunslinger!
 - “Shoot first, then relax later”



Boundary-value and eigenvalue problems

- Iterative shooting procedure :
 - Guess unknown initial parameter
 - Generate trial solution with a “marching” algorithm
 - Compute difference at boundary
 - Iterate until difference is small
 - Use a root-finding method



Boundary-value and eigenvalue problems

- Iterative shooting procedure :
 - Guess unknown initial parameter
 - Generate trial solution with a “marching” algorithm
 - Compute difference at boundary
 - Iterate until difference is small
 - Use a root-finding method!

```
class BVPStep( BVP ) :
    def __init__( self, x0, x1, u0, u1, delta, N ) :
        BVP.__init__( self, x0, x1, u0, u1, delta, N )
    def __call__( self, xy ): # derivative vector
        x = xy[0] ; u = xy[1] ; uprime = xy[2]
        dx_dx = 1.0
        du_dx = uprime
        duprime_dx = - math.pi**2 * (u + 1) / 4
        return [ dx_dx, du_dx, duprime_dx ]
    def step( self ) :
        cpt.RK4_step( self.xy, self.dx, self )

class BVPShoot( BVP ) :
    step = None
    def __init__( self, step, x0, x1, u0, u1, delta, N ) :
        BVP.__init__( self, x0, x1, u0, u1, delta, N )
        self.step = step
    def __call__( self, x ) : # function whose root is to be found
        self.delta = x
        self.xy = [ self.x0, self.u0, self.delta ]
        for i in range( self.N ) :
            cpt.RK4_step( self.xy, self.dx, self.step )
        return self.xy[1] - self.u1
    def solve( self ) :
        return cpt.root_secant( self, self.delta, self.delta + self.dx )
```

Boundary-value and eigenvalue problems

```
print " Steping solution of u'' = - pi^2 (u+1) / 4"
x0, x1, u0, u1 = input( "Input x0, x1, u0, u1 : ")
delta = input( "Input initial guess : ")

stepper = bvp.BVPStep( x0=x0, x1=x1, u0=u0, u1=u1, delta=delta, N= 100)
shooter = bvp.BVPShoot( step=stepper, x0=x0, x1=x1, u0=u0, u1=u1, delta=delta, N= 100)

slope = shooter.solve()
print ''
print " Shooting method found slope = ", slope
print ""
print ""
x_plot = []
u_plot = []
#print " Trajectory : "
stepper.xy = [ x0, u0, slope ]
#print '{0:>8s} {1:>8s} {2:>8s}'.format( 'x', 'u', "u'" )
#print '{0:8.4f} {1:8.4f} {2:8.4f}'.format( stepper.xy[0], stepper.xy[1], stepper.xy[2] )
x_plot.append( stepper.xy[0] )
u_plot.append( stepper.xy[1] )
for i in range(stepper.N):
    stepper.step()
    #print '{0:8.4f} {1:8.4f} {2:8.4f}'.format( stepper.xy[0], stepper.xy[1], stepper.xy[2] )
    x_plot.append( stepper.xy[0] )
    u_plot.append( stepper.xy[1] )

import matplotlib
matplotlib.rcParams['legend.fancybox'] = True
import matplotlib.pyplot as plt

plt.plot( x_plot, u_plot )

plt.show()
```

Boundary-value and eigenvalue problems

- Iterative relaxation procedure :
 - Guess solution at all values of x AND the boundary
 - Compute the difference in the ODE

$$G(x) = \frac{d^2 u_g}{dx^2} - f(x, u_g, u'_g) ,$$

- Iterate adjustments until $G(x)$ tends to zero

Boundary-value and eigenvalue problems

- Specifically we use Jacobi's relaxation algorithm :
 - Discretize the space, compute second derivative:

$$h = \frac{1}{N}, \quad x_i = ih, \quad u_i \equiv u(x_i), \quad i = 1, \dots, N-1$$

$$\left. \frac{d^2 u}{dx^2} \right|_{x=x_i} = \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2}, \quad i = 1, \dots, N-1$$

- Compute difference:

$$G(x_i) \approx \frac{u_{g,i+1} + u_{g,i-1} - 2u_i}{h^2} - f(x_i, u_{g,i}, u'_{g,i}) \approx 0$$

- Solve for the next guess :

$$u_i \approx \frac{1}{2} \left[u_{g,i+1} + u_{g,i-1} + h^2 f(x_i, u_{g,i}, u'_{g,i}) \right]$$

- Like the Euler algorithm, Jacobi method will be the “workhorse” for many more advanced algorithms

Boundary-value and eigenvalue problems

- For eigenvalue problems, we've already seen this once

- Recall :
$$\frac{d^2u}{dx^2} = -\frac{\pi^2}{4}(u + 1) ,$$

with $u(0) = 0, u(1) = 1$

- We discretized this :

$$\frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = \frac{\pi^2}{4}(u_i + 1) , \quad i = 1, \dots, N - 1$$

- And put this into the matrix form:

$$\mathbf{M}\mathbf{u} = \mathbf{b} ,$$

Boundary-value and eigenvalue problems

- Specifically we use Jacobi's relaxation algorithm :
 - Discretize the space, compute second derivative:
 - Compute difference:
 - Solve for the next guess

```
relaxer = BVPRelax( x0=0., x1=1., u0=0., u1=1., N=100 )
relaxer.guess()
relaxer.relax()
iteration = 1
while relaxer.error() > acc:
    relaxer.relax()
    iteration += 1
    print " %6d      %18.16g  %18.16g" % (iteration, relaxer.change(), relaxer.error())

file = open("relaxing.data", "w")
for i in range(relaxer.N+1):
    x = relaxer.x0 + i * relaxer.dx
    file.write(repr(x) + "\t" + repr(relaxer.u[i]) + "\n")
file.close()
print " N =", relaxer.N, "step solution in file relaxing.data"
```


Boundary-value and eigenvalue problems

```
class BVPRelax( BVP ) :
    u = None
    uOld = None
    def __init__( self, x0, x1, u0, u1, N ) :
        BVP.__init__( self, x0, x1, u0, u1, 0.0, N )
        self.u = [0.0] * (N+1)
        self.uOld = [0.0] * (N+1)

    def g(self, u, x):          # RHS function
        return math.pi**2 / 4 * (u + 1)

    def uExact(self, x):       # exact solution
        return math.cos(math.pi * x / 2) + 2 * math.sin(math.pi * x / 2) - 1

    def guess(self):           # linear fit to boundary conditions
        for i in range(self.N+1):
            x = self.x0 + i * self.dx
            self.u[i] = self.u0 + (x - self.x0) / (self.x1 - self.x0) * (self.u1 - self.u0)

    def relax(self):           # Jacobi algorithm
        for i in range(self.N+1):
            self.uOld[i] = self.u[i] # save old values
        for i in range(1, self.N):
            x = self.x0 + i * self.dx
            self.u[i] = 0.5 * ( self.uOld[i+1] + self.uOld[i-1] + self.dx**2 * self.g(self.uOld[i], x) )

    def change(self):          # compute maximum change
        maxDiff = 0.
        for i in range(1, self.N):
            diff = abs(self.u[i] - self.uOld[i])
            if diff > maxDiff:
                maxDiff = diff
        return maxDiff

    def error(self):           # compute maximum change
        maxDiff = 0.
        for i in range(1, self.N):
            x = self.x0 + i * self.dx
            diff = abs(self.u[i] - self.uExact(x))
            if diff > maxDiff:
                maxDiff = diff
        return maxDiff
```


Boundary-value and eigenvalue problems

- We write M as a tridiagonal matrix :

$$\mathbf{M} = \begin{pmatrix} 2-c & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2-c & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2-c & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2-c & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2-c \end{pmatrix},$$

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} u_0 + c \\ c \\ c \\ \vdots \\ c \\ u_N + c \end{pmatrix},$$

- You played with this in your homework, so we won't belabor the point, but realize that this is intricately tied!

Boundary-value and eigenvalue problems

- Can also solve for eigenvalues by adjusting the parameters until the boundary conditions are met
- In this case we know $\psi(x=0)$ and $\psi(x=1)$ are both equal to zero
- So, adjust energy until this occurs!

Boundary-value and eigenvalue problems

```
class ParticleInBox :
    E = 0.0          # current value of E for root finding
    accuracy = 1.0e-6 # accuracy for adaptive RK4 integration
    def __init__( self, E, accuracy ) :
        self.E = E
        self.accuracy = accuracy

class ParticleInBoxEnergy( ParticleInBox ) :
    def __init__( self, E, accuracy ) :
        ParticleInBox.__init__( self, E, accuracy )
    def __call__(self, xy): # time independent Schroedinger equation
        x = xy[0] ; psi = xy[1] ; psi_prime = xy[2]
        dx_dx = 1.0
        dpsi_dx = psi_prime
        dpsi_prime_dx = ( self.V(x) - self.E ) * psi
        return [ dx_dx, dpsi_dx, dpsi_prime_dx ]
    def V(self, x): # potential function
        if x >= 0.0 and x <= 1.0:
            return 0.0
        else:
            return 1.0e30

class ParticleInBoxWavefunction( ParticleInBox ) :
    def __init__( self, E, accuracy ) :
        ParticleInBox.__init__( self, E, accuracy )
        self.particleE = ParticleInBoxEnergy( E, accuracy )
    def __call__(self, energy): # search function for root finding
        self.E = energy
        self.particleE.E = energy
        x = 0.0 # left wall
        psi = 0.0 # psi(0)
        psi_prime = 1.0 # arbitrary value of slope
        xy = [ x, psi, psi_prime ]
        Delta_x = 1.0 # width of well
        dx = 0.001 # suggested step size of Runge-Kutta
        cpt.RK4_integrate(xy, dx, self.particleE, Delta_x, self.accuracy)
        return xy[1] # searching for psi(x=1) = 0

print " Bound state energies of infinitely deep potential well"
print " -----"
E0, E1, accuracy = input(" Enter guess E_0, dE, and desired accuracy: ")
particlePsi = ParticleInBoxWavefunction( E0, accuracy )
E = cpt.root_bisection(particlePsi, E0, E1, particlePsi.accuracy, 1000, True)
print " Eigenvalue =", E
```

Boundary-value and eigenvalue problems

- To continue our investigation of BVP and eigenvalue problems, consider

$$\frac{d^2 u}{dx^2} + d(x) \frac{du}{dx} + q(x)u = s(x) ,$$

–where $d(x)$, $q(x)$ and $s(x)$ are given functions

- The $s(x)$ term makes the equation inhomogeneous
- The Sturm-Liouville theory deals with linear homogeneous second order equations of the form

$$-\frac{d}{dx} \left[p(x) \frac{du}{dx} \right] + r(x)u(x) = \lambda w(x)u(x) ,$$

–where $p(x)$, $r(x)$, $w(x)$ are given, and λ is a parameter, $p(x) > 0$ and $w(x) > 0$ in integration domain

- Need boundary conditions! Consider homogenous and linear BC's like :

$$c_1 u(a) + c_2 u'(a) = 0 , \quad c_3 u(b) + c_4 u'(b) = 0 ,$$

Boundary-value and eigenvalue problems

- Sturm and Liouville showed :
 - Non-trivial solutions exist only for eigenvalues λ
 - If eigenvalues are arranged in increasing order, eigenfunctions have one additional node or zero per step
- Can solve these types of equations with the Numerov's Method
- If we have a second-order ODE without a first-order derivative term : $\frac{d^2u}{dx^2} + q(x)u(x) = s(x)$,
- Then the symmetric three-point difference is :

$$\frac{u_{n+1} + u_{n-1} - 2u_n}{h^2} = u_n'' + \frac{h^2}{12}u_n'''' + \mathcal{O}(h^4) .$$

Boundary-value and eigenvalue problems

- With the differential equation, we can write :

$$\begin{aligned} u_n'''' &= \frac{d^2}{dx^2} \left(-q(x)u(x) + s(x) \right) \Big|_{x=x_n} \\ &= -\frac{q_{n+1}u_{n+1} - 2q_n u_n + q_{n-1}u_{n-1}}{h^2} + \frac{s_{n+1} - 2s_n + s_{n-1}}{h^2} + \mathcal{O}(h^2) . \end{aligned}$$

- If we plug this into the difference formula and simplify we get:

$$\begin{aligned} &\left(1 + \frac{h^2}{12}q_{n+1} \right) u_{n+1} - 2 \left(1 + \frac{5h^2}{12}q_n \right) u_n + \left(1 + \frac{h^2}{12}q_{n-1} \right) u_{n-1} \\ &= \frac{h^2}{12} \left(s_{n+1} + 10s_n + s_{n-1} \right) + \mathcal{O}(h^6) . \end{aligned}$$

- Already better than RK4!
- But, this is a three-point formula, so needs u_0 and u_1 to start it

Boundary-value and eigenvalue problems

- Now we consider the Quantum harmonic oscillator

- Hamiltonian is :

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2\hat{x}^2,$$

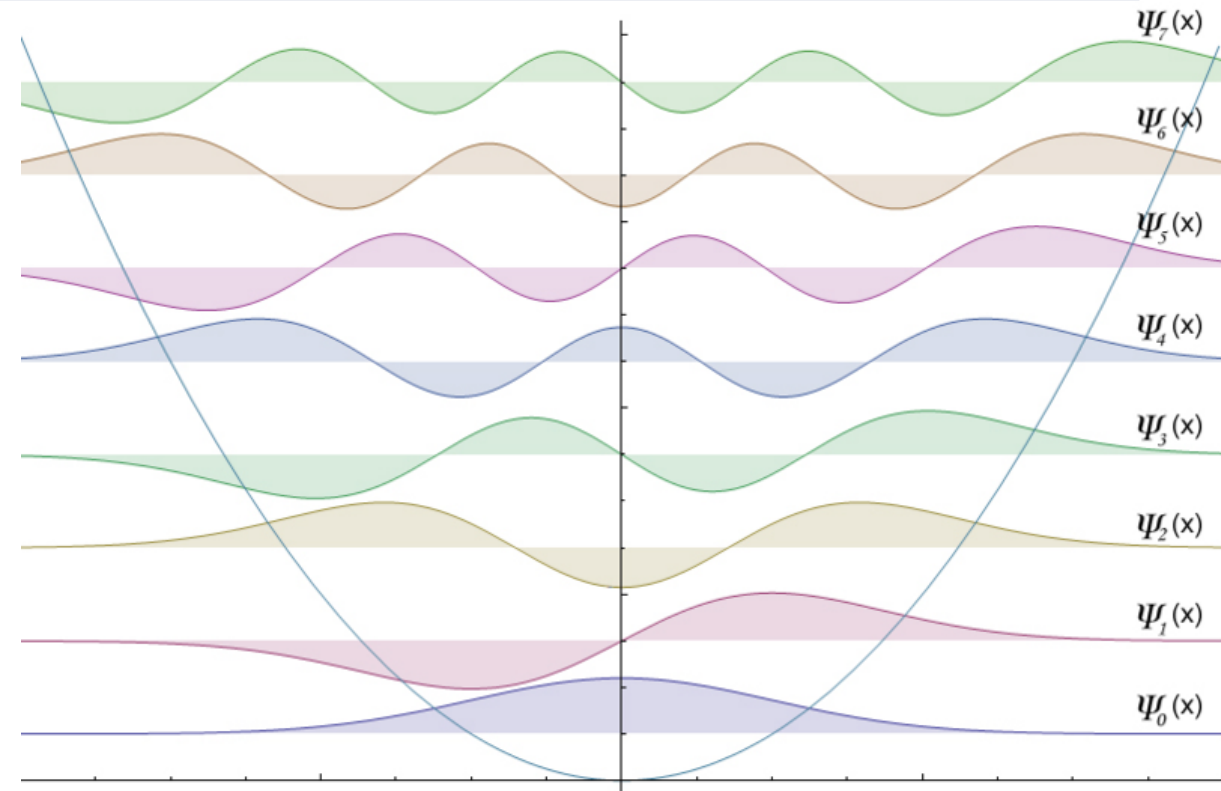
- Eigenfunctions are :

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \cdot \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \cdot e^{-\frac{m\omega x^2}{2\hbar}} \cdot H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right), \quad n = 0, 1, 2, \dots$$

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} (e^{-x^2})$$

- Energy levels are :

$$E_n = \hbar\omega \left(n + \frac{1}{2}\right)$$



Boundary-value and eigenvalue problems

- Obviously can't compute for x in \pm infinity
 - Need to get appropriate boundary conditions
 - Choose left and right boundaries that are “big enough” and then apply approximate boundary conditions :
 - Since $\psi(x) \sim 0$ there, just set $\psi(x) = 0$
- We can use the Numerov algorithm to solve this
- Some caveats :
 - There are unphysical solutions that march to infinity (mathematical property of QHO)
 - From symmetry, need to make sure that the solutions are appropriately symmetric or antisymmetric!

Boundary-value and eigenvalue problems

- How to deal with this?
 - March twice!
 - Once from left
 - Once from right
 - Ensure that they match at some x value
 - We can actually multiply one of the solutions by a constant (still solves the ODE) so we ensure

$$\phi_{\text{left}}(x_{\text{match}}) = \phi_{\text{right}}(x_{\text{match}}) .$$

- If we have a true match, then we can test

$$\left. \frac{d\phi_{\text{left}}}{dx} \right|_{x_{\text{match}}} = \left. \frac{d\phi_{\text{right}}}{dx} \right|_{x_{\text{match}}} .$$

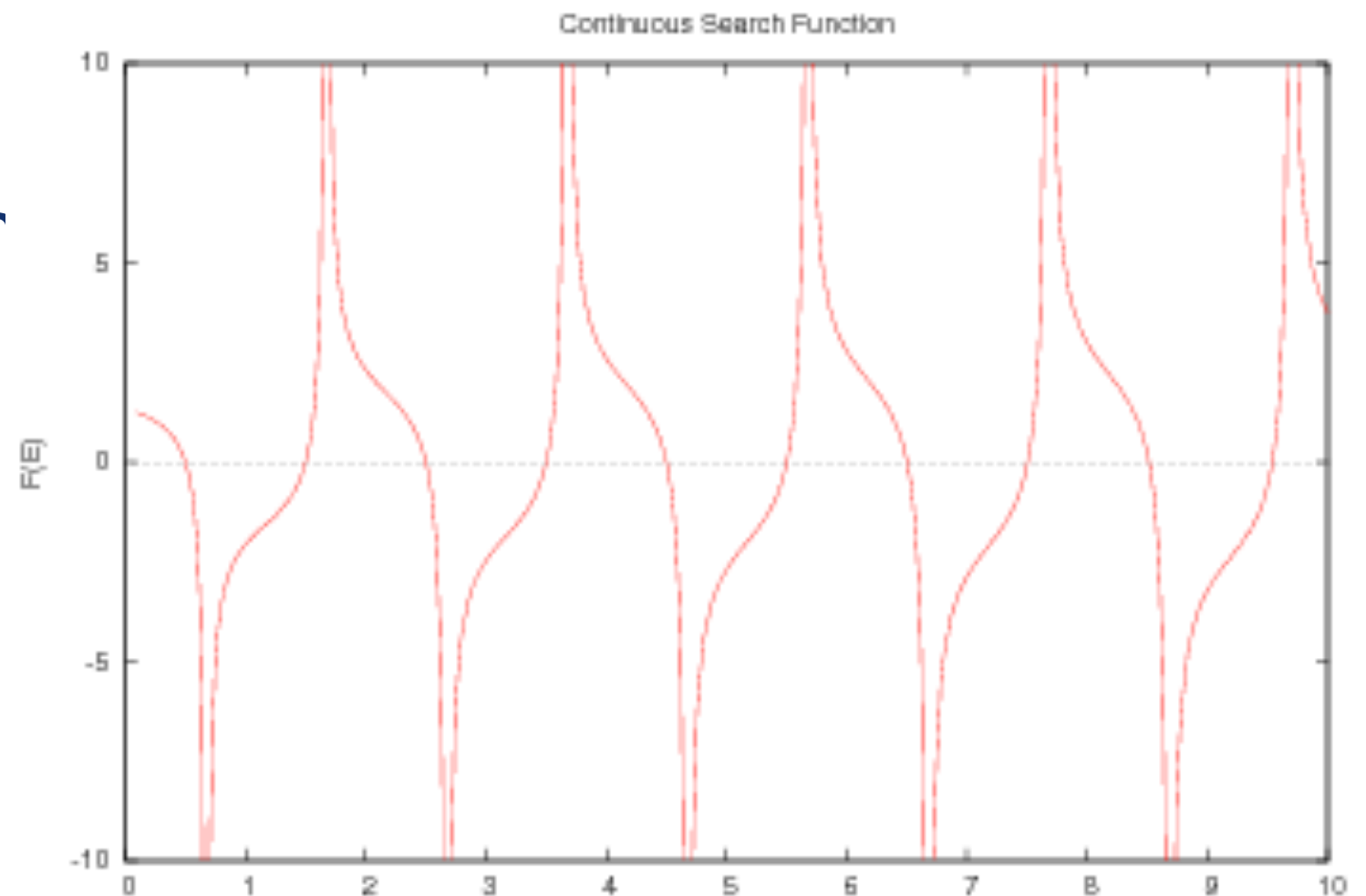
Boundary-value and eigenvalue problems

- Can pick a matching point near the classical turning point with $E = V(x)$

- Matching condition is :

$$F(E) = \pm [\phi_{\text{left}}(x_{\text{match}} + h) - \phi_{\text{left}}(x_{\text{match}} - h)] \\ - [\phi_{\text{right}}(x_{\text{match}} + h) - \phi_{\text{right}}(x_{\text{match}} - h)] / (2h\phi(x_{\text{match}})) \\ = 0 .$$

- Finally, we can find the matching function $F(E)$ numerically and find the roots!



Code for Schroedinger

Find the roots for the matching condition

Find the roots for the “q” function (eigenvalues)

```
while True:                                # loop over levels

    # estimate next E and dE
    dE = 0.5 * (schroedinger.E - E_old)
    E_old = schroedinger.E
    schroedinger.E += dE

    # use simple search to locate root with relatively low accuracy
    accuracy = 0.01
    schroedinger.E = cpt.root_simple(schroedinger.F, schroedinger.E, dE, accuracy)
    #simple_steps = cpt.root_steps()

    # use secant search with relatively high accuracy
    accuracy = 1.0e-6
    E1 = schroedinger.E + 100 * accuracy # guess second point required
    schroedinger.E = cpt.root_secant(schroedinger.F, schroedinger.E, E1, accuracy)
    #secant_steps = cpt.root_steps()

    #ans = " " + repr(level).rjust(3) + " "*5 + repr(E).ljust(20) + " "*6
    #ans += repr(simple_steps).rjust(3) + " "*11 + repr(secant_steps).rjust(3)
    #print ans
    level += 1

    accuracy = 0.001
    x = cpt.root_simple(schroedinger.q, schroedinger.x_left, schroedinger.h, accuracy)
    swrite = '{0:12.6f} {1:12.6f}'.format( x, schroedinger.E )
    print swrite

    x = cpt.root_simple(schroedinger.q, schroedinger.x_right, -schroedinger.h, accuracy)
    swrite = '{0:12.6f} {1:12.6f}'.format( x, schroedinger.E )
    print swrite

    print ''

    schroedinger.normalize()
    x_data.append( [] )
    phi_data.append( [] )
    iphi = len(x_data) - 1
    for i in range(schroedinger.N+1):
        x = schroedinger.x_left + i * schroedinger.h
        x_data[iphi].append(x)
        phi_data[iphi].append( schroedinger.phi[i] )

    if schroedinger.E >= E_max:                # we are done
```

Code for Schroedinger

Integrate from the left

Integrate from the right

Make sure they match up

Be sure to flip the sign
every time a new node happens

```
def F(self, energy):
    # eigenvalue at F(E) = 0

    # set energy needed by the q(x) function
    self.E = energy

    # find the right turning point
    i_match = self.N
    x = self.x_right
    # start at right boundary
    while self.V(x) > self.E:
        # in forbidden region
        i_match -= 1
        x -= self.h
        if i_match < 0:
            raise Exception("can't find right turning point")

    # integrate self.phi_left using Numerov algorithm
    self.phi_left[0] = 0.0
    self.phi_left[1] = 1.0e-10
    c = self.h**2 / 12.0
    # constant in Numerov formula
    for i in range(1, i_match+1):
        x = self.x_left + i * self.h
        self.phi_left[i+1] = 2 * (1 - 5 * c * self.q(x)) * self.phi_left[i]
        self.phi_left[i+1] -= (1 + c * self.q(x - self.h)) * self.phi_left[i-1]
        self.phi_left[i+1] /= 1 + c * self.q(x + self.h)

    # integrate self.phi_right
    self.phi[self.N] = self.phi_right[self.N] = 0.0
    self.phi[self.N-1] = self.phi_right[self.N-1] = 1.0e-10
    for i in range(self.N - 1, i_match - 1, -1):
        x = self.x_right - i * self.h
        self.phi_right[i-1] = 2 * (1 - 5 * c * self.q(x)) * self.phi_right[i]
        self.phi_right[i-1] -= (1 + c * self.q(x + self.h)) * self.phi_right[i+1]
        self.phi_right[i-1] /= 1 + c * self.q(x - self.h)
        self.phi[i-1] = self.phi_right[i-1]

    # rescale self.phi_left
    scale = self.phi_right[i_match] / self.phi_left[i_match]
    for i in range(i_match + 2):
        self.phi_left[i] *= scale
        self.phi[i] = self.phi_left[i]

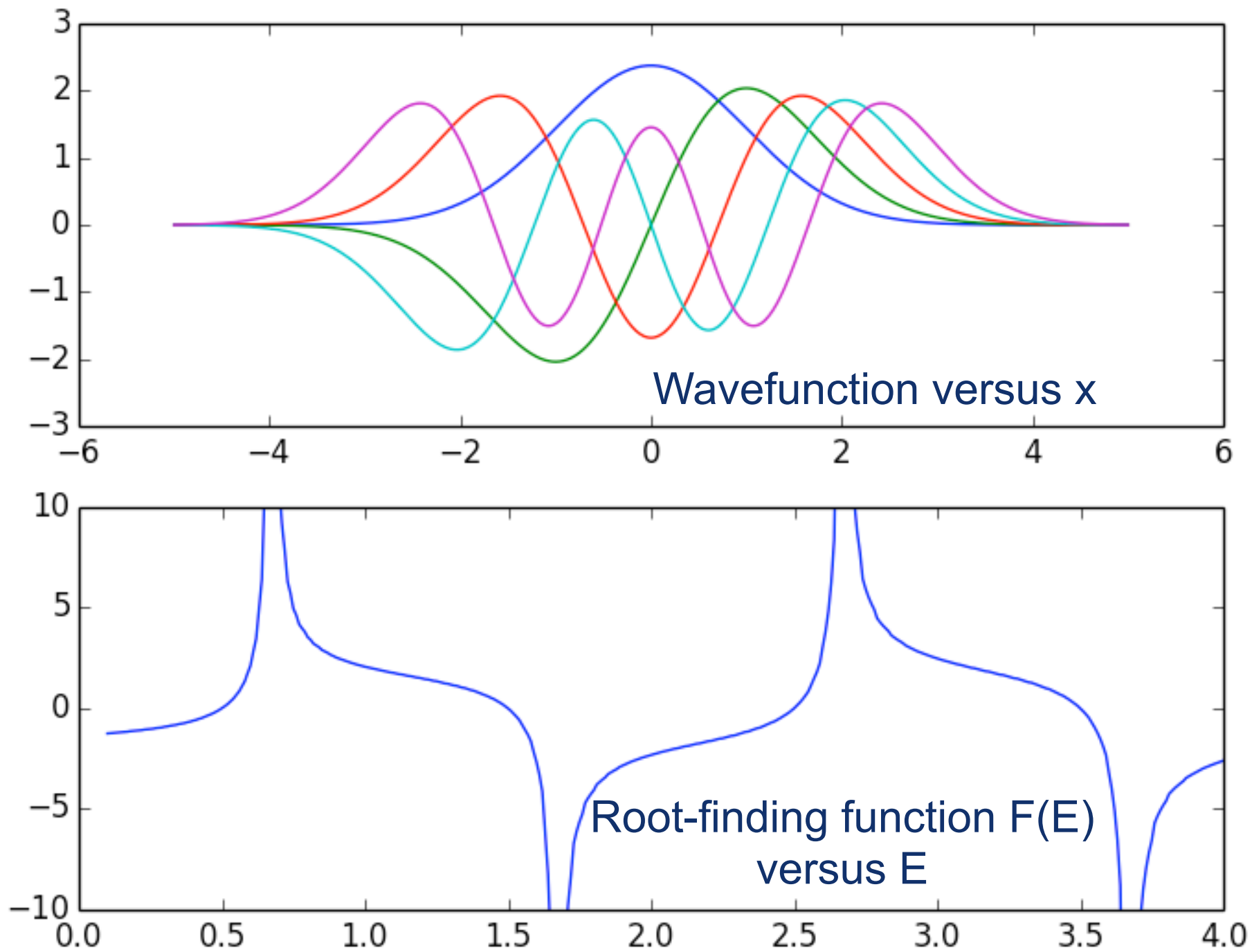
    # make F(E) continuous
    # count number of nodes in self.phi_left
    n = 0
    for i in range(1, i_match+1):
        if self.phi_left[i-1] * self.phi_left[i] < 0.0:
            n += 1

    # flip its sign when a new node develops

    if n != self.nodes:
        self.nodes = n
        self.sign = -self.sign

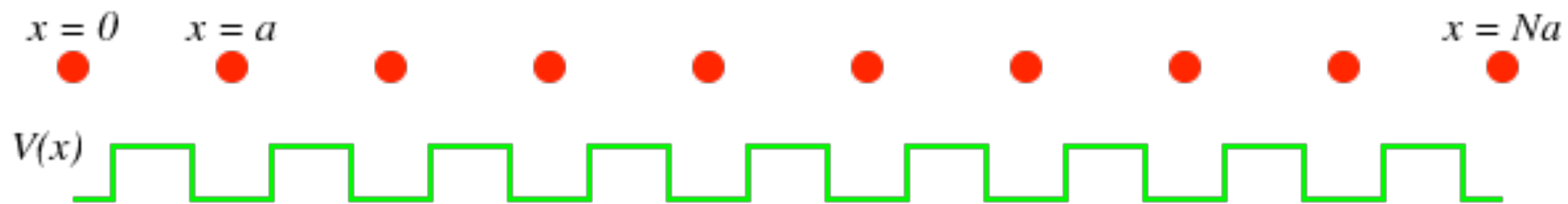
    return ( self.sign *
            ( self.phi_right[i_match-1] - self.phi_right[i_match+1] -
              self.phi_left [i_match-1] + self.phi_left[i_match+1] ) /
            (2 * self.h * self.phi_right[i_match]) )
```

Results of Schroedinger



Boundary-value and eigenvalue problems

- Another popular problem is to investigate Particle in a periodic potential
 - Kronig-Penney model : 1-d “crystal”
 - N heavy nuclei at fixed lattice sites with spacing a



- This is easier to solve in the Fourier domain!

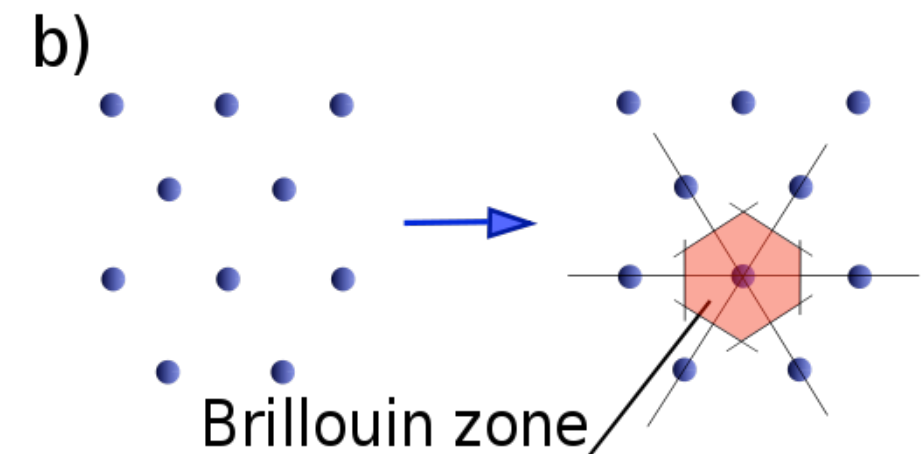
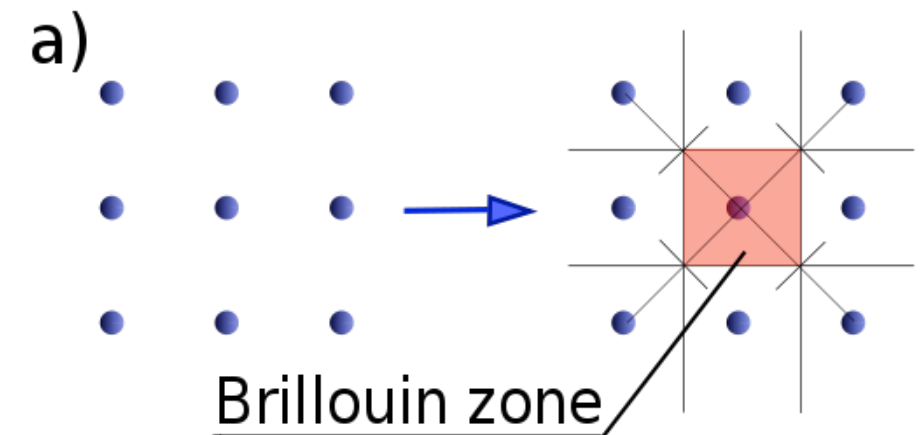
$$f(x) = \sum_k e^{ikx} f_k ,$$

- Then we have periodic boundary conditions :

$$e^{ik(x+Na)} = e^{ikx} \quad \Rightarrow \quad k = \frac{2\pi n}{Na} , \text{ where } n = 0, \pm 1, \pm 2, \dots$$

Boundary-value and eigenvalue problems

- Define the Brillouin zone :
 - http://en.wikipedia.org/wiki/Brillouin_zone
- Space of nearest neighbors!
- Also known as the Voronoi cell
 - http://en.wikipedia.org/wiki/Voronoi_cell



Boundary-value and eigenvalue problems

- Define the lattice point by translating the objects by some basis vector \mathbf{a} , and the “reciprocal” basis vectors \mathbf{b} :

$$\mathbf{R} = n\mathbf{a} , \text{ where } \mathbf{a} = a\hat{x} , \text{ and } n = 0, \pm 1, \pm 2, \dots ,$$

$$\mathbf{a} \cdot \mathbf{b} = 2\pi \quad \text{that is} \quad \mathbf{b} = (2\pi/a)\hat{x} .$$

$$\mathbf{K} = n\mathbf{b} , \text{ where } n = 0, \pm 1, \pm 2, \dots$$

- The first Brillouin zone is :

$$k = 0, \pm \frac{2\pi}{Na}, \pm \frac{4\pi}{Na}, \pm \frac{6\pi}{Na}, \dots, \pm \frac{(N-1)\pi}{Na}, \frac{\pi}{a} .$$

- A general wave number q can be decomposed into a wave number k in the first Brillouin zone and a wave number of the reciprocal lattice $q = k + K$.

Boundary-value and eigenvalue problems

- Can use “Bloch’s theorem” to solve the problem

- We know:
$$V(x) = \sum_K e^{iKx} V_K ,$$

- The eigenstates satisfy:
$$-\frac{\hbar^2}{2m} \frac{d^2 \psi}{dx^2} + V(x)\psi(x) = E\psi(x) .$$

- Then we expand in Fourier series:

$$\psi(x) = \sum_q e^{iqx} C_q = \sum_{K,k} e^{i(k+K)x} C_{k+K} ,$$

- Plugging this into Schroedinger’s equation we get:

$$\sum_{K,k} \left\{ \left[\frac{1}{2}(k+K)^2 - E \right] e^{i(k+K)x} C_{k+K} \right\} + \sum_{K'} V_{K'} \sum_{K,k} e^{i(k+K+K')x} C_{k+K} = 0 .$$
$$\hbar^2 / m = 1$$

Boundary-value and eigenvalue problems

- k and k' are both reciprocal lattice vectors, so we just change the notation:

$$\sum_{K,k} e^{i(k+K)x} \left\{ \left[\frac{1}{2} (k+K)^2 - E \right] C_{k+K} + \sum_{K'} V_{K-K'} C_{k+K'} \right\} = 0 .$$

- The functions are linearly independent so :

$$\left[\frac{1}{2} (k+K)^2 - E \right] C_{k+K} + \sum_{K'} V_{K-K'} C_{k+K'} = 0 .$$

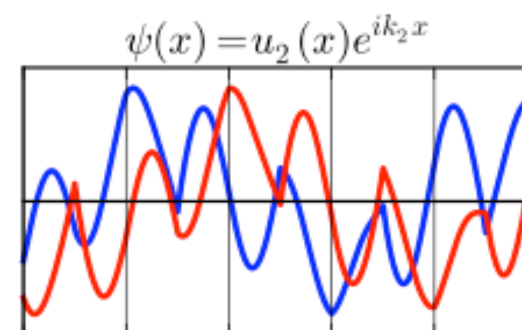
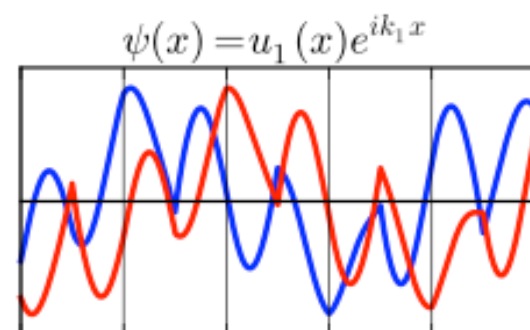
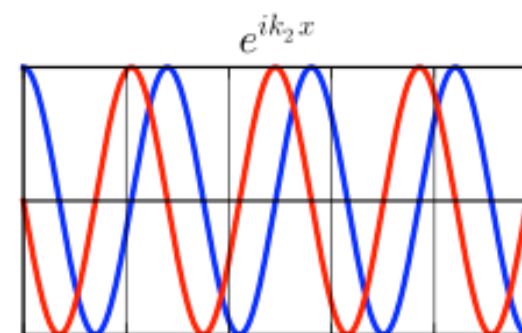
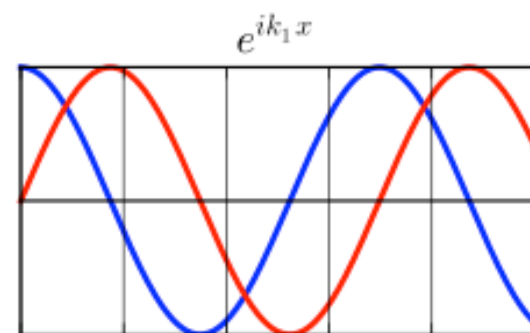
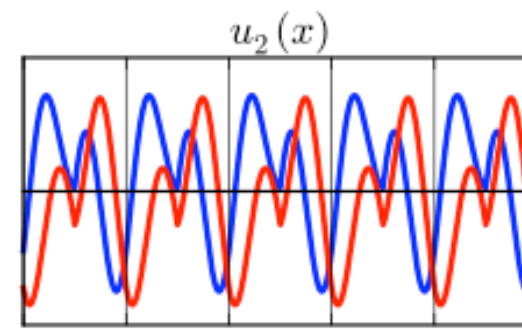
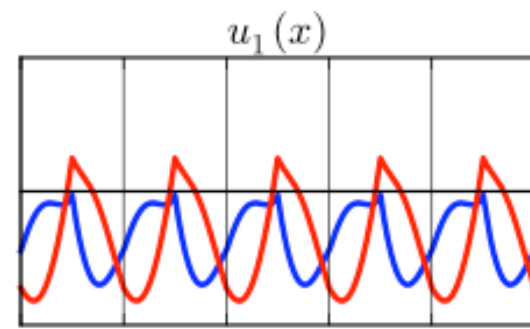
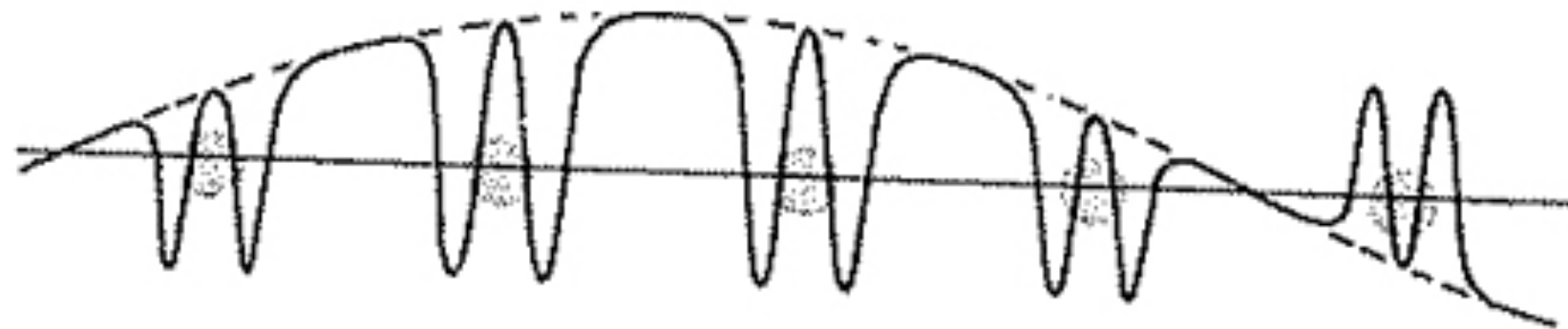
- Modes with different wave numbers k **decouple** from one another!
- Can state this as: $\psi_k(x) = e^{ikx} \sum_K e^{iKx} C_{k+K} = e^{ikx} u_k(x)$,
– where :

$$u_k(x + na) = u_k(x) ,$$

- This is Bloch's theorem!

Boundary-value and eigenvalue problems

- So this “feels like” the solution to the first Brillouin zone, shifted by a simple plane wave!



Boundary-value and eigenvalue problems

- To solve for the band structure we look at the eigenvalue equation :

$$\left[\frac{1}{2} (k + K)^2 - E \right] C_{k+K} + \sum_{K'} V_{K-K'} C_{k+K'} = 0 .$$

- Actually infinite-dimensional, so we need to cut it off somewhere
- Looking at the spectrum of states as a function of k over the first Brillouin zone is a “band”

Boundary-value and eigenvalue problems

- Concretely, consider a single cell on the lattice :

$$V(x) = \begin{cases} 0 & \text{for } -\frac{a}{2} < x < -\frac{\Delta}{2} \\ V_0 & \text{for } -\frac{\Delta}{2} < x < \frac{\Delta}{2} \\ 0 & \text{for } \frac{\Delta}{2} < x < \frac{a}{2} \end{cases} ,$$

- Between barriers we have

$$\psi(x) = A_n e^{iq(x-na)} + B_n e^{-iq(x-na)} ,$$

- Inside the barrier we have

$$\psi(x) = C_n e^{i\kappa(x-na)} + D_n e^{-i\kappa(x-na)} ,$$

- Here : $q = \sqrt{2E}$ $\kappa = \sqrt{2(E - V_0)}$

- Determine A,B,C,D's by matching psi and psi', then we have recursion relation to solve for n+1 given n:

$$\begin{pmatrix} A_{n+1} \\ B_{n+1} \end{pmatrix} = \mathbf{T}(E) \begin{pmatrix} A_n \\ B_n \end{pmatrix} ,$$

Boundary-value and eigenvalue problems

- $T(E)$ is the “transfer matrix”, with elements :

$$T_{11} = T_{22}^* = \frac{e^{iq(a-\Delta)}}{4q\kappa} \left[e^{i\kappa\Delta} (q + \kappa)^2 - e^{-i\kappa\Delta} (q - \kappa)^2 \right] ,$$
$$T_{12} = T_{21}^* = -\frac{ie^{iq(a-\Delta)}}{2q\kappa} (q^2 - \kappa^2) \sin(\kappa\Delta) .$$

- From Bloch’s theorem, eigenvalues are of the form $\exp(ika)$, where k is the reduced wave number in the first Brillouin zone
- $T(E)$ is 2x2 so can easily solve the characteristic equation:

$$\det [\mathbf{T} - e^{ika} \mathbf{1}] = 0 .$$

- This yields a quadratic equation with two solutions for $k(E)$ equal in magnitude and opposite in sign

Code for Kronig-Penney

Get the transfer matrix

Solve for eigenvalues
with characteristic equation

Bands are the k-values
versus energy

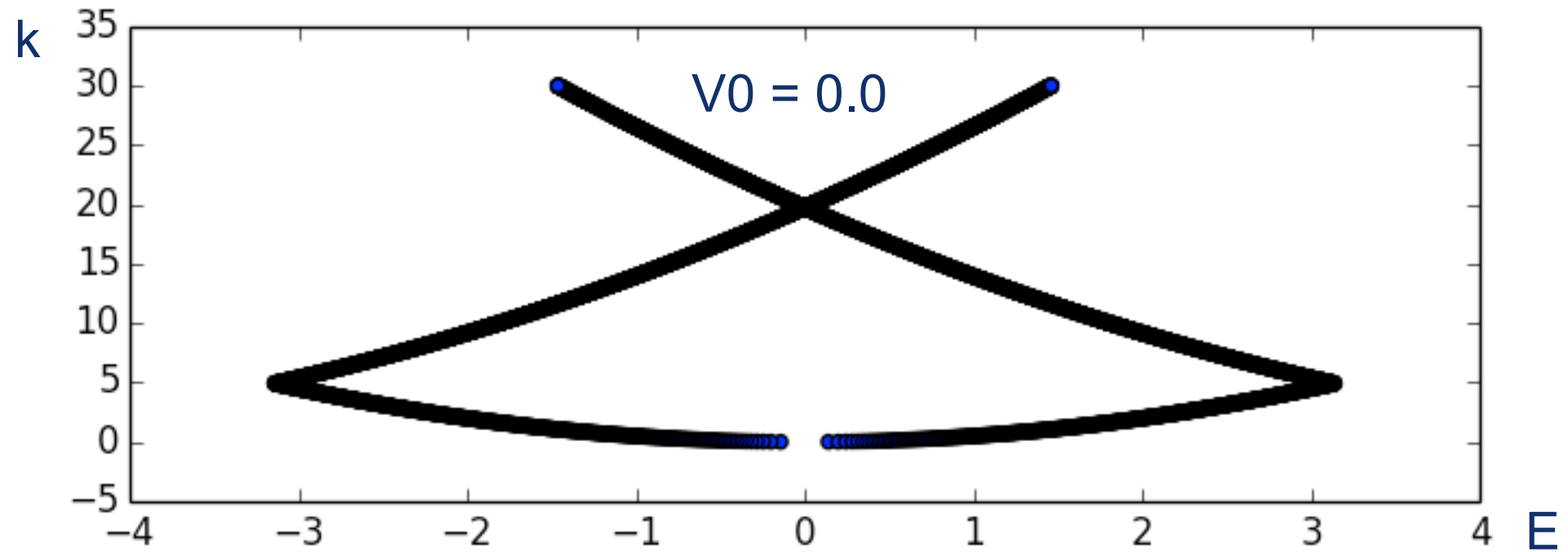
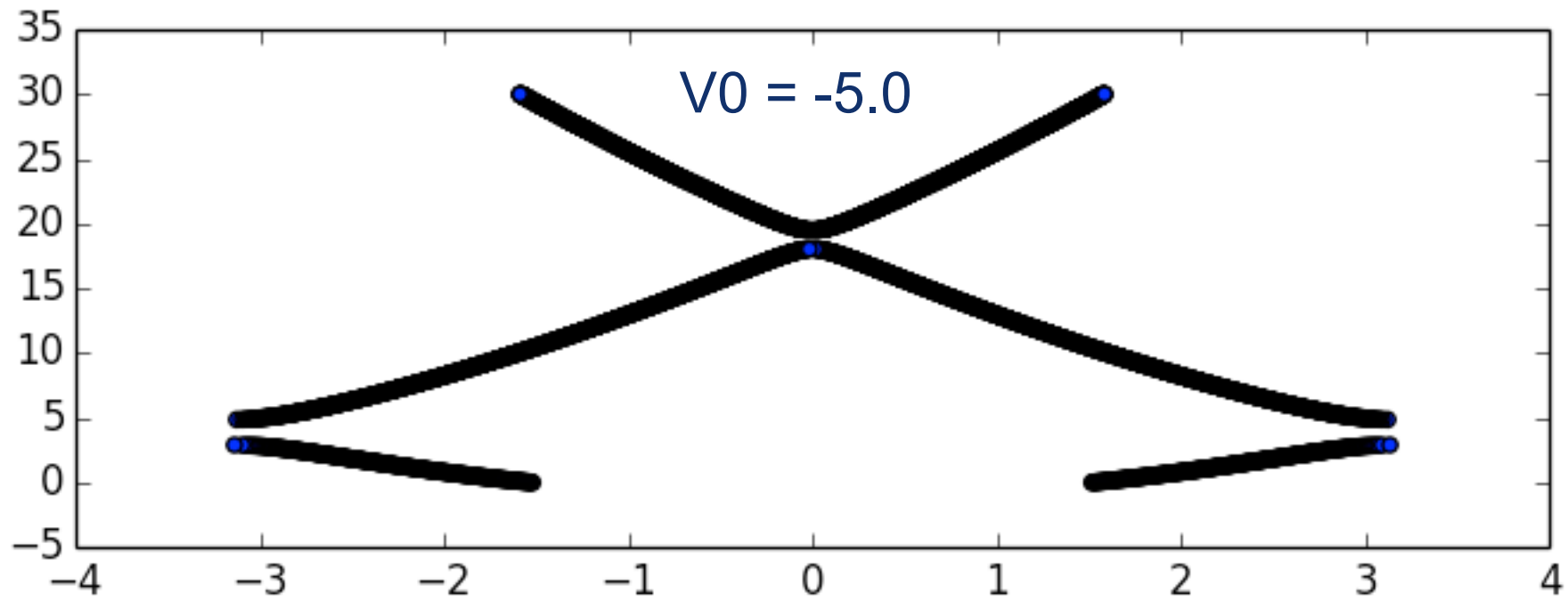
```
class KronigPenney :
    def __init__(self) :
        self.a = 1.0 # size of unit cell - lattice spacing
        self.V_0 = -5.0 # height of potential barrier
        self.Delta = 0.2 # width of potential barrier

    def solve_for_E(self, E, k): # to solve 2x2 eigenvalue problem
        # E is the desired energy (input)
        # k is a list of the two solutions
        q = math.sqrt(2 * E)
        kappa = math.sqrt(2 * (E - self.V_0))
        i = 1.0j
        T11 = ( cmath.exp(i * q * (self.a - self.Delta)) / (4 * q * kappa) *
              ( cmath.exp(i * kappa * self.Delta) * (q + kappa)**2 -
                cmath.exp(-i * kappa * self.Delta) * (q - kappa)**2 ) )
        T22 = T11.conjugate()
        T12 = ( -i * cmath.exp(i * q * (self.a - self.Delta)) / (2 * q * kappa) *
              (q**2 - kappa**2) * math.sin(kappa * self.Delta) )
        T21 = T12.conjugate()

        # solve quadratic determinantal equation
        b = - (T11 + T22)
        c = (T11 * T22 - T12 * T21)
        k[0] = (- b + cmath.sqrt(b**2 - 4*c)) / 2.0
        k[1] = (- b - cmath.sqrt(b**2 - 4*c)) / 2.0
        for j in range(2):
            k[j] = cmath.log(k[j]) / (i * self.a)

    def compute_bands(self, dE, steps, band_file_name):
        # dE = step size in E for search
        # steps = number of steps
        E_values = []
        rq_values = []
        file = open(band_file_name, "w")
        E = dE
        for step in range(steps):
            q = [ 0.0 + 0.0j, 0.0 + 0.0j ]
            self.solve_for_E(E, q)
            for j in range(2):
                rq = q[j].real
                if rq > 0.0 and rq < math.pi / self.a:
                    rq_values.append( rq )
                    rq_values.append( -rq )
                    E_values.append( E )
                    E_values.append( E )
            E += dE
        return rq_values, E_values
```


Results of Kronig-Penney



Results of Kronig-Penney

