Series of Quantum Finite Potential Wells as an Energy State Approximation of Solids

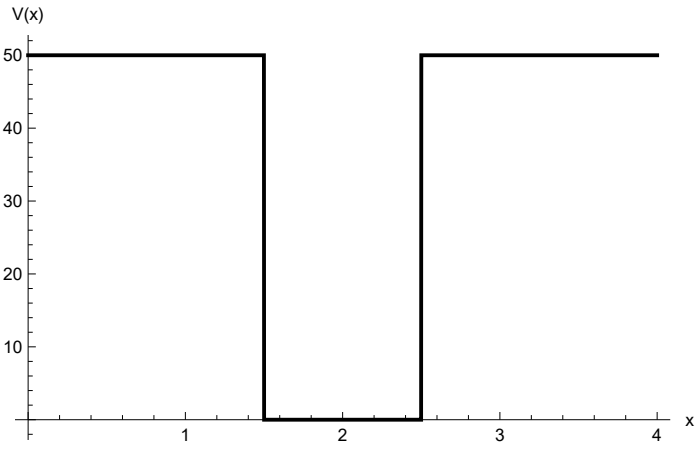
Finite Potential Well

Shooting Method

First, we begin by defining a potential well with width 1, depth 50, and centered at x=2 for subsequent calculations.

In[1]:=
```
Lx = 1;
xCenter = 2;
v[x_] := If[xCenter - Lx / 2 < x < xCenter + Lx / 2, 0, 50]
xMin = 0;
xMax = 4;
vPlot = Plot[v[x], {x, xMin, xMax},
  Exclusions → None, AxesLabel → {"x", "V(x)"}, PlotStyle → Black]
```

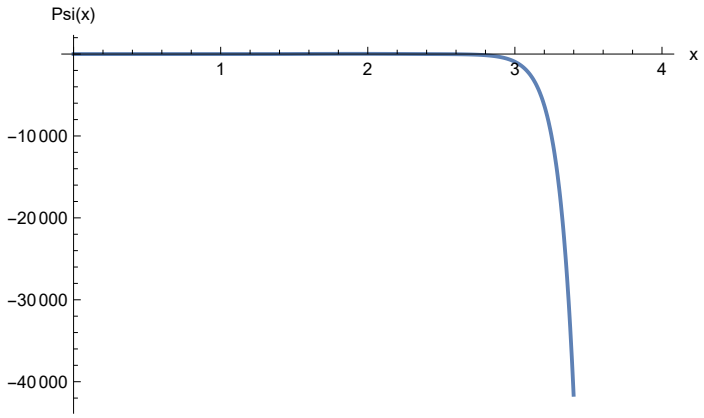
Out[6]=



Then, we use the shooting method to determine the wavefunction which solves the TISE. First, we try using E=5. Note that the derivative is set to a small value 0.0001 because a derivative value of 0 would lead to a flat wavefunction (trivial).

In[7]:=
```
m = 1;
h = 1;
energy = 5;
f[x_] = -2 m (energy - v[x]) / h^2;
solution = NDSolve[
   {psi''[x] == f[x] × psi[x], psi[xMin] == 0, psi'[xMin] == 0.0001}, psi, {x, xMin, xMax}];
ShootingPlot = Plot[psi[x] /. solution, {x, 0, 4}, AxesLabel → {"x", "Psi(x)"}]
```

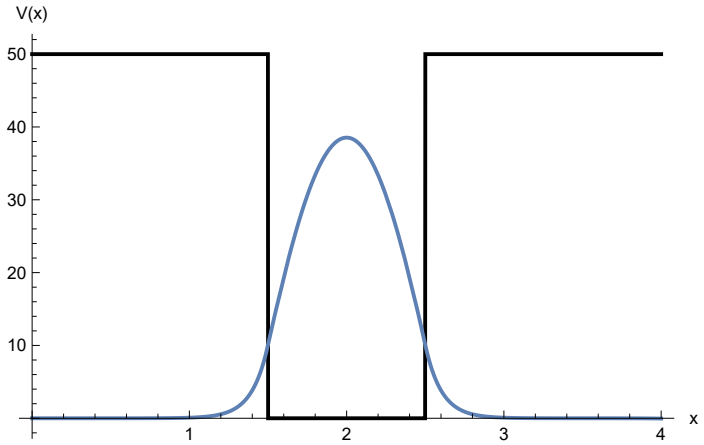
Out[12]=



Now, we converge to a reasonable solution (where the function smoothly decays at the potential wells and there are 0 nodes - corresponds to ground state).

In[13]:=
```
m = 1;
h = 1;
energy = 3.413571;
f[x_] = -2 m (energy - v[x]) / h^2;
solution = NDSolve[
   {psi''[x] == f[x] × psi[x], psi[xMin] == 0, psi'[xMin] == 0.0001}, psi, {x, xMin, xMax}];
SPlot = Plot[psi[x] /. solution, {x, 0, 4}, AxesLabel → {"x", "Psi(x)"}];
Show[vPlot, SPlot, PlotRange → All, PlotLegends → {"V(x)", "Psi(x)"}]
```

Out[19]=

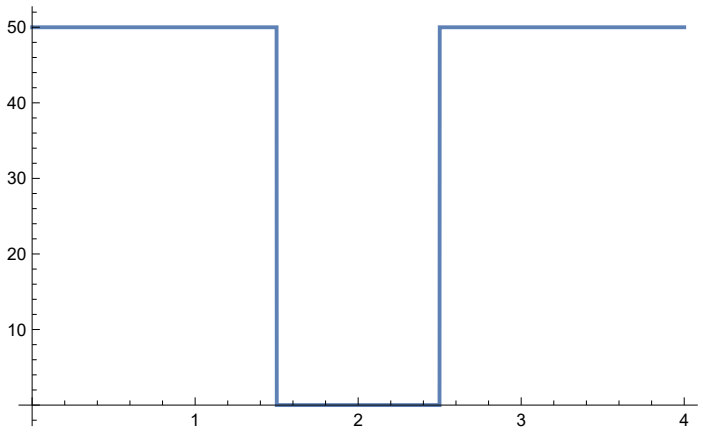


Matrix Diagonalization Method

After, we use the matrix diagonalization technique to solve the TISE more efficiently. Again, we start by defining the potential.

In[20]:=
```
b = 4;
v[x_] := If[Abs[x - b / 2] < 0.5, 0, 50];
Plot[v[x], {x, 0, b}, Exclusions → None]
```

Out[22]=



Next, we define the basis function phi(x) and the kinetic energy matrix using the derived infinite potential well energy states (E=n^2pi^2/(2mb^2), where m, the mass, is 1 and b is the length of the infinite potential well). Lastly, we create a tridiagonal matrix and solve using the sparse matrix method Eigensystem[].

In[23]:=
```
nMax = 50;
phi[n_, x_] := Sqrt[2 / b] * Sin[n Pi x / b];
vMatrix =
  Table[NIntegrate[phi[n, x] × v[x] × phi[m, x], {x, 0, b}], {n, 1, nMax}, {m, 1, nMax}];
h0Matrix = DiagonalMatrix[
   Table[(n^2 Pi^2 / (2 b^2)) KroneckerDelta[m, n], {n, 1, nMax}, {m, 1, nMax}]];
hmnMatrix = h0Matrix + vMatrix;
{eValues, eVectors} = Eigensystem[hmnMatrix];
```

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {1.4999999730972267602774999659743696651600930458414495660690590739 3}. NIntegrate obtained -5.32907×10^{-15} and 4.924999093114498`*^-12 for the integral and error estimates.

NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {2.93064}. NIntegrate obtained 4.8433479449272454`*^-15 and 2.5223218153881677`*^-15 for the integral and error estimates.

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {2.5000000269027732397225000340256303348399069541585504339309409260 7}. NIntegrate obtained 3.608224830031759`*^-15 and 4.569487644531984`*^-12 for the integral and error estimates.

General: Further output of NIntegrate::ncvb will be suppressed during this calculation.

NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

General: Further output of NIntegrate::slwcon will be suppressed during this calculation.

NIntegrate: Catastrophic loss of precision in the global error estimate due to insufficient WorkingPrecision or divergent integral.

NIntegrate: Catastrophic loss of precision in the global error estimate due to insufficient WorkingPrecision or divergent integral.

NIntegrate: Catastrophic loss of precision in the global error estimate due to insufficient WorkingPrecision or divergent integral.

General: Further output of NIntegrate::errprec will be suppressed during this calculation.

NIntegrate: Integral and error estimates are 0 on all integration subregions. Try increasing the value of the MinRecursion option. If value of integral may be 0, specify a finite value for the AccuracyGoal option.

NIntegrate: Integral and error estimates are 0 on all integration subregions. Try increasing the value of the MinRecursion option. If value of integral may be 0, specify a finite value for the AccuracyGoal option.

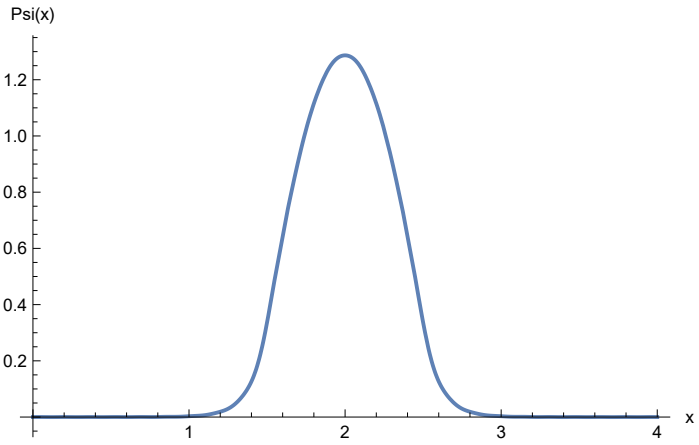
NIntegrate: Integral and error estimates are 0 on all integration subregions. Try increasing the value of the MinRecursion option. If value of integral may be 0, specify a finite value for the AccuracyGoal option.

General: Further output of NIntegrate::izero will be suppressed during this calculation.

Then, we plot the wavefunction calculated from the matrix diagonalization technique below. Note how similar this ground state looks to the shooting method and has an energy state that is identical.

In[29]:=
```
mPlot1 = Plot[-Sum[eVectors[[50, n]] * phi[n, x], {n, 1, nMax}],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
eValues[[50]]
```

Out[29]=



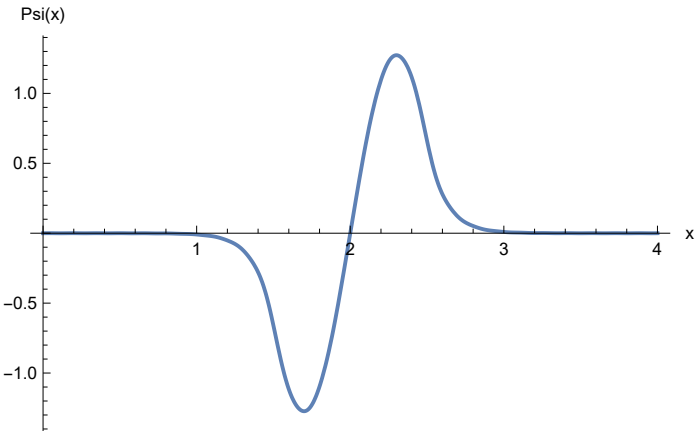
Out[30]=

```
3.41566
```

Lastly to prove the superiority of this method to shooting, we plot the first excited state and its associated energy value.

In[31]:=
```
mPlot2 = Plot[-Sum[eVectors[[49, n]] * phi[n, x], {n, 1, nMax}],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
eValues[[49]]
```

Out[31]=



Out[32]=

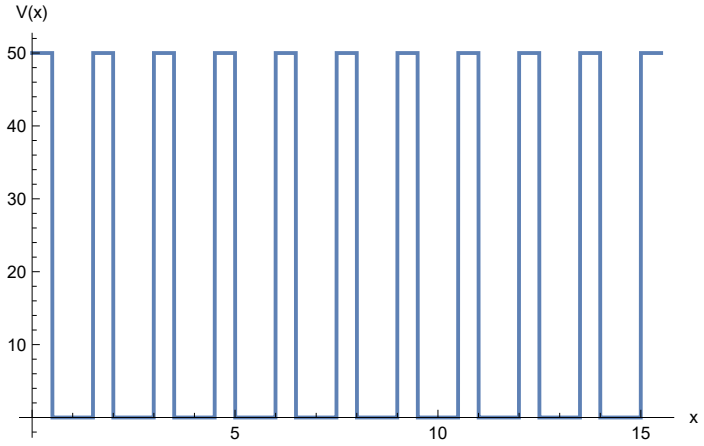
```
13.4824
```

Repeating Potential Wells

The next step we take is repeating the finite potential wells. We begin by defining the corresponding potential for 10 wells. Note that b corresponds to the maximum x value. A neat trick we can do is create a pattern by using the modulus operator for the current position and the well width. This takes the fractional quotient between the two. Because this is predictably less than 1 in the desired potential well region we can use this to define multiple wells of width bWidth (Schroeder, 2022).

In[33]:=
```
nWells = 10;
bWidth = 0.5;
b = nWells + (nWells + 1) bWidth;
v[x_] := If[Mod[x - bWidth, 1 + bWidth] < 1, 0, 50]
Plot[v[x], {x, 0, b}, Exclusions → None, AxesLabel → {"x", "V(x)"}]
```

Out[37]=



Next, we solve the corresponding eigenvalue problem just as before.

In[38]:=
```
nMax = 25;
phi[n_, x_] := Sqrt[2 / b] * Sin[n Pi x / b];
vMatrix =
  Table[NIntegrate[phi[n, x] × v[x] × phi[m, x], {x, 0, b}], {n, 1, nMax}, {m, 1, nMax}];
h0Matrix = DiagonalMatrix[
   Table[(n^2 Pi^2 / (2 b^2)) KroneckerDelta[m, n], {n, 1, nMax}, {m, 1, nMax}]];
hmnMatrix = h0Matrix + vMatrix;
{eValues, eVectors} = Eigensystem[hmnMatrix];
```

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {4.99999991032408920092499988658123221720031015280483188689686357 98}. NIntegrate obtained -5.82867×10^{-16} and 4.745342063568309`*^-13 for the integral and error estimates.

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {9.49999999103240892009249998865812322172003101528048318868968635 80}. NIntegrate obtained -9.99201×10^{-16} and 5.76519528164256`*^-13 for the integral and error estimates.

NIntegrate: NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} = {9.00000000896759107990750001134187677827996898471951681131031364202}. NIntegrate obtained -3.08781×10^{-16} and 6.017624745222388`*^-13 for the integral and error estimates.

General: Further output of NIntegrate::ncvb will be suppressed during this calculation.

NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

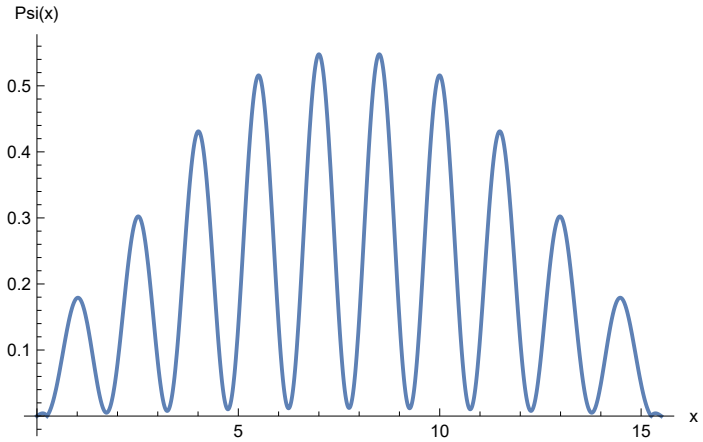
NIntegrate: Numerical integration converging too slowly; suspect one of the following: singularity, value of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

General: Further output of NIntegrate::slwcon will be suppressed during this calculation.

After, we plot the probability density of the first 4 energy states. Note how they correspond to linear combinations of constructive and destructive interference.

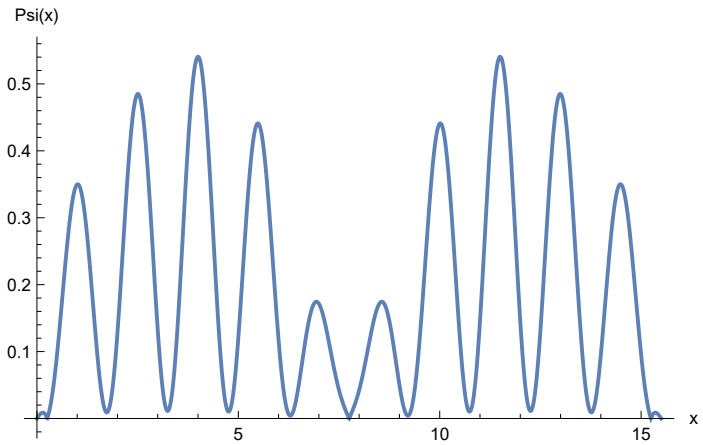
In[44]:=
```
muPlot1 = Plot[Abs[Sum[eVectors[[25, n]] * phi[n, x], {n, 1, nMax}]],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
```

Out[44]=



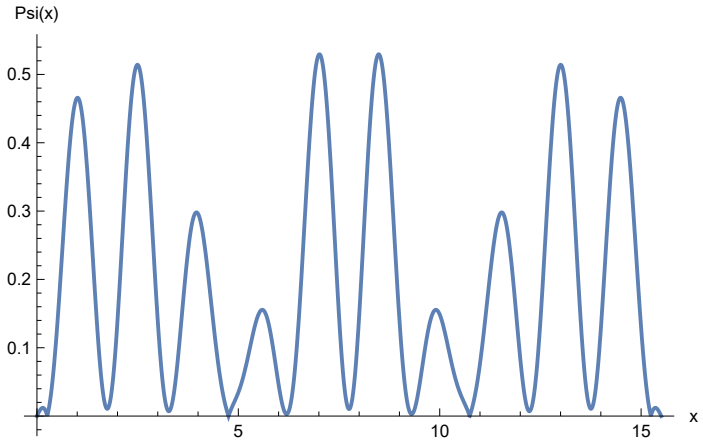
In[45]:=
```
muPlot2 = Plot[Abs[Sum[eVectors[[24, n]] * phi[n, x], {n, 1, nMax}]],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
```

Out[45]=



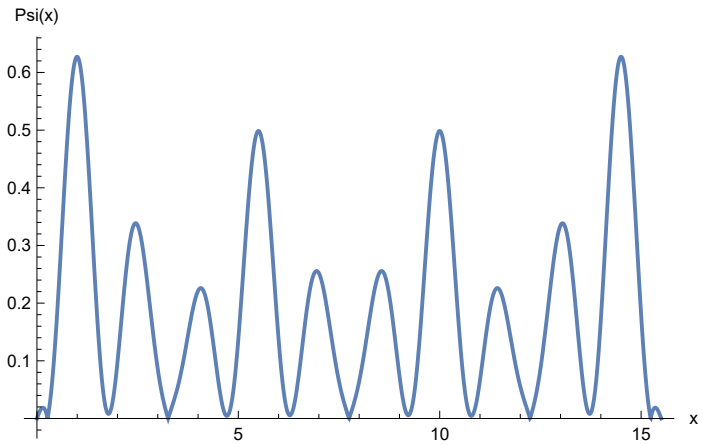
In[46]:=
```
muPlot3 = Plot[Abs[Sum[eVectors[[23, n]] * phi[n, x], {n, 1, nMax}]],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
```

Out[46]=



In[47]:=
```
muPlot4 = Plot[Abs[Sum[eVectors[[22, n]] * phi[n, x], {n, 1, nMax}]],
  {x, 0, b}, PlotRange → All, AxesLabel → {"x", "Psi(x)"}]
```

Out[47]=



Last but not least we can plot the band structure plot or the eigenvalues corresponding to the various linear combinations of atomic orbitals (LCAO). This is where we make the connection between the LCAO method being an approximation of band structure in solids like crystals.

In[48]:=
```
eValuesc = Select[eValues, # < 45 &];
ListPlot[Transpose[{eValuesc, ConstantArray[1, Length[eValuesc]]}],
 Filling → Axis, FillingStyle → Directive[Opacity[0.5], Red],
 PlotStyle → Vertical, PlotRange → All, AxesLabel → {"Energy"}]
```

Out[49]=