```
(*Poisson solver that implements the first part of
 the algorithm described below, solving over full space,
including periodic edge - it represents the Laplacian as a matrix operator then
  inverts it. Any solution to Poisson's equation is now found by multiplying
  the inverse matrix by the (forcing function ρ? idk if you can call it that)*)
(*This is a symmetric finite difference method to solve the Poisson Equation*)
(*en.wikipedia.org/wiki/Discrete_Poisson_equation*)
PoissonSolver[MM_, NN_, chargeList_List, ϵ1_, ϵ2_, H_, a_, λ_] := Module[
  {m, m1, m2, m3, m4, m5, m6, m7, m8, m9, newm, Ry, α, γ, plotList,
   totalCharge, tempTotal, list, potential, exteriorPoints, chargeVector},

  (*construct D, the matrix operator that corresponds to the
   Laplacian. This is a 9 point stencil. It's periodic with respect to two
   edges: Born-von Karman boundaries. One edge is fixed to 0 everywhere:
    Dirichlet boundaries. The last edge has the normal component
     of the flux at the boundary: Neumann boundaries.*)
  periodicblock[c_, b_] := c IdentityMatrix[MM - 1] + DiagonalMatrix[
      Table[b, {i, 0, MM - 3}], 1] + DiagonalMatrix[Table[b, {i, 0, MM - 3}], -1] +
    DiagonalMatrix[{b}, -MM + 2] + DiagonalMatrix[{b}, MM - 2];
  m1 = periodicblock[-2./3, -1/6];
  m2 = periodicblock[10/3, -2/3];
  m3 = KroneckerProduct[DiagonalMatrix[Join[Table[1, {i, 1, NN}], {0}], 0], m2];

  m4 = KroneckerProduct[DiagonalMatrix[Join[Table[1, {i, 1, NN - 1}], {0}], 1], m1];
  m5 =
   KroneckerProduct[DiagonalMatrix[Join[Table[1, {i, 1, NN - 1}], {0}], -1], m1];
  m6 = KroneckerProduct[DiagonalMatrix[{1}, NN], m1];
  m7 = 1/2 IdentityMatrix[MM - 1];
  (*The matrix has been augmented so that the bottom blocks
   correspond to a row of "ghost" points above the top row. THey
   are used to establish a centered second order approximation
   for the first derivative on V normal to the boundary. This *)
  m8 = KroneckerProduct[DiagonalMatrix[{0, 1}, -NN + 1], m7];
  m9 = KroneckerProduct[DiagonalMatrix[Join[Table[0, {i, 1, NN}], {1}], 0], -m7];
  m = m3 + m4 + m5 + m6 + m8 + m9;
  newm = m3 + m4 + m5;


  α = .529/a; (*a0/a, where a0 is the Bohr radius*)
  Ry = 13.60569253; (*Rydberg unit of energy, in eV. 8πRy=e^2/(a0ϵ0) in eV*)
```

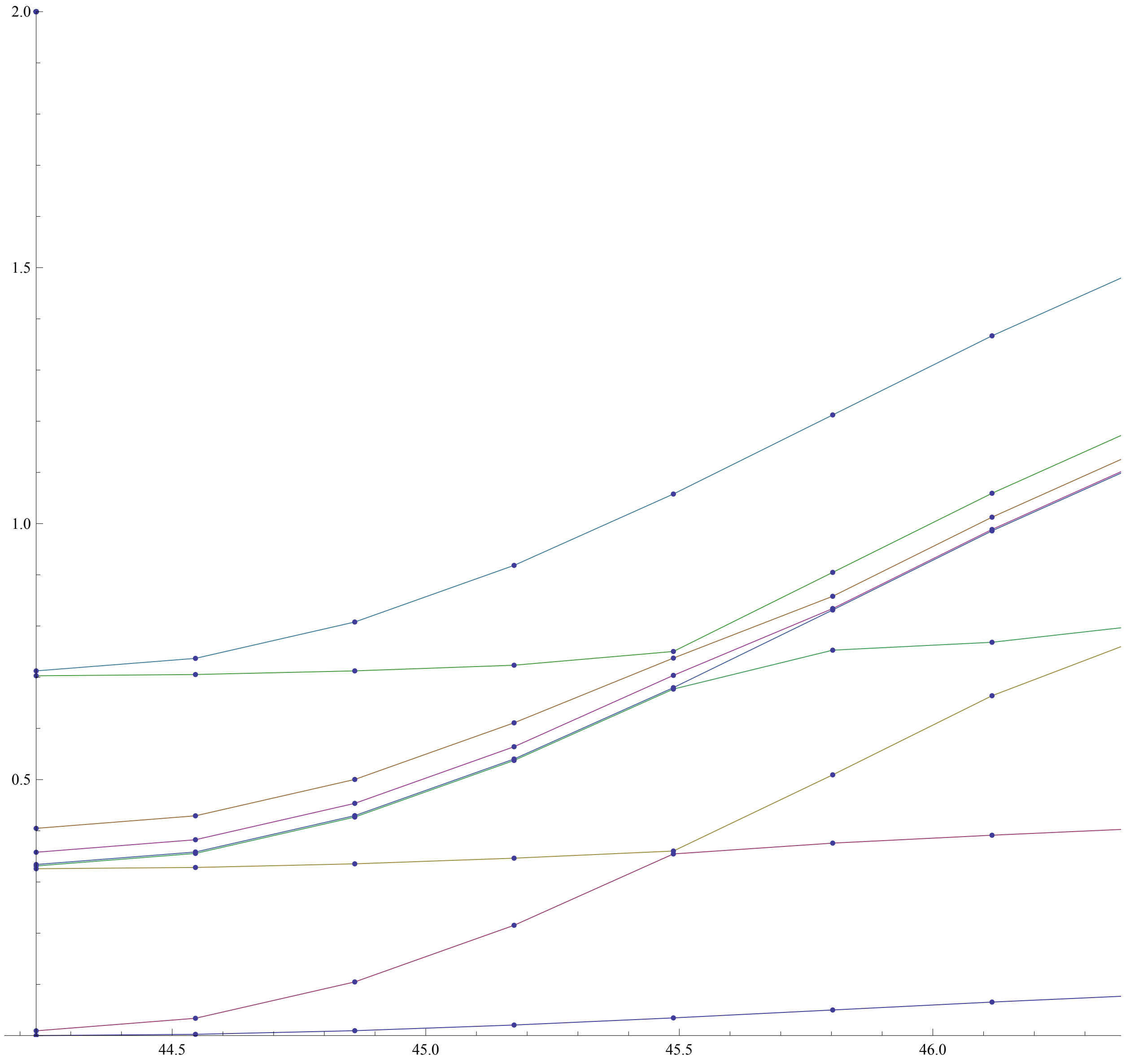
```
totalCharge = -(ϵ2/ϵ1) (λ/N[π]) (1/2 ArcTan[(MM a)/(2 H)] - 1/2 ArcTan[-((MM - 2) a)/(2 H)]);
(*analytic expression for the total charge*)


tempTotal = Flatten[chargeList] // Total;
(*coefficient in front of n on the right hand side of the poisson equation*)
γ = -(8 N[π] Ry α)/ϵ2;
e1[j_] := (8 N[π] Ry α λ H a)/(2 N[π] ϵ1 (H^2 + (j - Floor[(MM+1)/2])^2 a^2));
(*Normal component of electric field at top edge of the grid*)

(*Defines the list with the charge distribution ρij generated from the user,
augmented with the boundary conditions *)
chargeVector  = γ chargeList;
exteriorPoints =
 Table[
    N[e1[j ]] ,
   {j, 1, MM - 1}
  ] // Flatten;
chargeVector = Join[chargeVector, exteriorPoints];
chargeVector = chargeVector - 1/2 newm.chargeVector;
(*Calculates the potential*)
potential = LinearSolve[m, chargeVector];

(*Reindexes the potential and
 outputs a list of voltage formatted as {x,y,V(x,y)}*)
list = {};
Do[
 AppendTo[list,
  Table[
   potential[[j + (MM - 1) (i - 1)]],
   {j, 1, MM - 1}
  ]
 ];
 AppendTo[list, potential[[1 + (MM - 1) (i - 1)]]];
 list = list // Flatten;
 , {i, 1, NN}
];
Flatten[Table[
  {i, j, list[[j + (MM) (i - 1)]]},
  {i, 1, NN},
  {j, 1, MM}
 ], 1]
 ]
PoissonPlot[MM_, NN_, chargeList_List, ϵ1_, ϵ2_, H_, a_, λ_ ] :=
 Module[{plotList = PoissonSolver[MM, NN, chargeList, ϵ1, ϵ2, H, a, λ ]},
  ListPlot3D[plotList, PlotRange → All,
   PlotLabel → "Calculated Electric Potential", PlotRange → All]
 ]


(*gets rid of periodic edge, formats list for hamiltonian*)
TripleVoltage[MM_, NN_, chargeList_List, ϵ1_, ϵ2_, H_, a_, λ_ ] :=
 Module[{list = PoissonSolver[MM, NN, chargeList, ϵ1, ϵ2, H, a, λ][[All, 3]]},
  list = Drop[list, {MM, -1, MM}];
  Table[
    list[[j]],
    {j, 1, list // Length},
    {i, 1, 3}
   ] // Flatten
 ]

BaseMatrixTakesChargeList[MM_, NN_, chargeList_List, ϵ1_, ϵ2_, H_, a_, λ_, t1_,
  t2_ ] := Module[{list = TripleVoltage[MM, NN, chargeList, ϵ1, ϵ2, H, a, λ ],
   voltageMatrix, tx, ty, m1, m2, m3, m4},
  voltageMatrix = DiagonalMatrix[list];
  tx = DiagonalMatrix[{t2, t1, t1}, 0];
  ty = DiagonalMatrix[{t1, t2, t1}, 0];

  (*These 4 matrices are interatomic
    bonding in each of the 4 directions in the x-y plane*)
  m1 = KroneckerProduct[DiagonalMatrix[Table[1, {NN (MM - 1) - 1}], 1], ty];
  m2 = KroneckerProduct[DiagonalMatrix[Table[1, {NN (MM - 1) - 1}], -1], ty];
  m3 =
   KroneckerProduct[DiagonalMatrix[Table[1, {NN (MM - 1) - (MM - 1)}], (MM - 1)], tx];
  m4 = KroneckerProduct[DiagonalMatrix[
     Table[1, {NN (MM - 1) - (MM - 1)}], -(MM - 1)], tx];
  m1 + m2 + m3 + m4 + voltageMatrix
 ]

(*Returns eigensystem for matrix for a single k state*)
eigensystem[kz_] := Module[
   {Eyz, Ezx, Exy, hamiltonian, m1, vals, vecs, degeneracyList},
   (*Eyz = 2t1 -2 t1 Cos[kz]+2t2+2t1;
   Ezx = 2t1 -2 t1 Cos[kz]+2t1+2t2;
   Exy = 2t2 -2 t2 Cos[kz]+2t1+2t1;*)
   Eyz = 2 t1 - 2 t1 Cos[kz];
   Ezx = 2 t1 - 2 t1 Cos[kz];
   Exy = 2 t2 - 2 t2 Cos[kz];
   hamiltonian = DiagonalMatrix[{Eyz, Ezx, Exy}, 0];
   m1 = KroneckerProduct[IdentityMatrix[NN * (MM - 1)], hamiltonian];
```

```
    matrix = baseMatrix + m1;
    {vals, vecs} = Eigensystem[matrix];
    degeneracyList = spinKDegeneracyFactor[kz] Table[1, {i, 1, vals // Length}];
    {vals, vecs, degeneracyList}
   ];

(*returns sorted list of energies for ALL k states*)
(*can parallelize later*)
TotalEnergySortedList[] :=
 Module[{vecs, vals, degeneracies, sys, orderList, temp},

   sys = eigensystem[0];
   vals = {sys[[1]]};
   vecs = {sys[[2]]};
   degeneracies = {sys[[3]]};

   SetSharedVariable[vals];
   SetSharedVariable[vecs];
   SetSharedVariable[degeneracies];

   ParallelDo[
    sys = eigensystem[i dk];
    AppendTo[vals, sys[[1]]];
    AppendTo[vecs, sys[[2]]];
    AppendTo[degeneracies, sys[[3]]];
    , {i, 1, numPartitions}
   ];

   vals = vals // Flatten;
   vecs = Flatten[vecs, 1];
   degeneracies = degeneracies // Flatten;

   orderList = Ordering[vals];
   vals = vals[[orderList]];
   vecs = vecs[[orderList]];
   degeneracies = degeneracies[[orderList]];

   {vals, vecs, degeneracies}
  ]

(*returns portion of the energy list
 to use and computers the fractional filling*)
NecessaryStates[numStatesTotal_] := Module[
    {list = TotalEnergySortedList[], sum = 0, count = 0, vals, vecs, degeneracies},
    degeneracies = list[[3]];
    Do[
     sum += degeneracies[[i]];
     If[sum > numStatesTotal,
      count = i;
      Break[];
     ];
    , {i, 1, degeneracies // Length}
   ];

   vals = list[[1]][[1 ;; count]];
   vecs = list[[2]][[1 ;; count]];
   degeneracies = degeneracies[[1 ;; count - 1]];

   degeneracies = Join[degeneracies, {numStatesTotal - (sum - list[[3, count]])}];

   If[sum - numStatesTotal == list[[3, count]], vals = Drop[vals, -1];
    vecs = Drop[vecs, -1]; degeneracies = Drop[degeneracies, -1]];
   EFermi = vals[[-1]];
   {vals, vecs, degeneracies}

  ];

(*α=1 for yz, 2 for zx, 3 for xy*)
orbitalDensity[n_, α_] := Module[{list = vecs[[n]][[α ;; ;; 3]]},
   Flatten[Table[
     {i , j , Norm[list[[j + (MM - 1) (i - 1)]]]^2},
     {i, 1, NN},
     {j, 1, MM - 1}
    ], 1]
  ];
totalDensity[n_] :=
  (orbitalDensity[n, 1] + orbitalDensity[n, 2] + orbitalDensity[n, 3])[[All, 3]];
(*α=1 for yz, 2 for zx, 3 for xy*)
totalOrbitallyProjectedDensity[n_, α_] := orbitalDensity[n, α][[All, 3]];
totalDensityPlot[n_] :=
  Module[{list = totalDensity[n], interp, table},
   table = Flatten[Table[
      {{i , j} , list[[j + (MM - 1) (i - 1)]]},
      {i, 1, NN},
      {j, 1, MM - 1}
     ], 1];
   interp = Interpolation[table, InterpolationOrder → 1];
   Plot3D[interp[x, y], {x, 1, NN}, {y, 1, MM - 1}, PlotRange → All]
  ];


spinKDegeneracyFactor[kz_] := 2 If[kz == 0 || kz == N[Pi], 1, 2];

(*returns CHARGE distribution, not density of states*)
normalizedSumList[numStatesTotal_] := Module[{totalList},
   {vals, vecs, degeneracies} = NecessaryStates[numStatesTotal];
```

```
    densityToNumStatesScaling
     ParallelSum[degeneracies[[n]] totalDensity[n], {n, 1, Length[vals]}]
   ];
 (*returns orbitally projected charge distribution*)
 normalizedProjectedSumList[numStatesTotal_, α_] := Module[{totalList},
    {vals, vecs, degeneracies} = NecessaryStates[numStatesTotal];
    densityToNumStatesScaling ParallelSum[
      degeneracies[[n]] totalOrbitallyProjectedDensity[n, α], {n, 1, Length[vals]}]
   ];

 normalizedSum[numStatesTotal_] := normalizedSumList[numStatesTotal] // Total;

 normalizedSumPlot[numStatesTotal_] :=
   Module[{list = normalizedSumList[numStatesTotal], interp, table},
    table = Flatten[Table[
        {{i , j} , list[[j + (MM - 1) (i - 1)]]},
        {i, 1, NN},
        {j, 1, MM - 1}
      ], 1];
    interp = Interpolation[table, InterpolationOrder → 1];
    Plot3D[interp[x, y], {x, 1, NN}, {y, 1, MM - 1}, PlotRange → All]
   ];

 BandStructure[minE_, maxE_] :=
  Module[{sys, plotList = {}, list2, length, FermiLine, f1, f2, shift},
   SetSharedVariable[plotList];
   Do[
    sys = eigensystem[i dk][[1]];
    If[i == 0, shift = sys[[1]]];
    length = sys // Length;
    AppendTo[plotList, Table[
      {i dk, sys[[j]]},
      {j, 1, length}
     ]];
    , {i, 0, numPartitions}
   ];
   list2 = Table[
     plotList[[i + 1]][[j]] - shift,
     {j, 1, length},
     {i, 0, numPartitions}
    ];
   FermiLine = Table[{i dk, EFermi}, {i, 0, numPartitions}];
   f1 = ListPlot[list2, Joined → True, Mesh → All, PlotRange → {minE, maxE}];
   f2 = ListPlot[FermiLine, Joined → True, PlotStyle → {Thick, Black}];
   Show[f1, f2]
  ]
```

In[863]:=
```
BandStructure[0, 2]
(*without parallel do*)
```

Out[863]=



2.0
1.5
1.0
0.5
44.5
45.0
45.5
46.0