Chemical Insight from Density Functional Modeling of Molecular Adsorption: Tracking the Bonding and Diffusion of Anthracene Derivatives on Cu(111) with Molecular Orbitals

Jonathan Wyrick,[1,a)] T. L. Einstein,[2] Ludwig Bartels[1,b)]

[1]Pierce Hall, University of California-Riverside, Riverside, California, 92521, USA

[2]Department of Physics and Condensed Matter Theory Center, University of Maryland, College Park, Maryland, 20742-4111, USA

[a)]Present address: National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, Maryland 20899
[b)]ludwig.bartels@ucr.edu

Mathematica Code for MO Diagram Generation

Contents

I. Initialization (MO Diagram Functions)

The primary function of interest is the *MOSpectrum* function, defined last in this section. All other functions shown here support the *MOSpectrum* function and are either called directly from it, or are to be used to populate data before calling the *MOSpectrum* function (*see section II for usage below*).

Set the working directory to be the location of this *Mathematica* notebook.

```
SetDirectory[NotebookDirectory[]];
```

▪ Function: Projections

result **= Projections[*dir*]**
This function parses the PROCAR file from the specified folder and returns an array containing the projections of each KS state onto the orbitals of each atom in the system. These are the KS functions projected onto the projector functions: $< \tilde{\psi}_m \mid \tilde{p}_{i^a} >$ as presented in the text.

Parameters
dir: the folder, relative to the working directory, in which the PROCAR file can be found.

Returns

result is an array containing the projections from PROCAR. The 1st index is the k - point index and the 2nd index references:
(index value = 1) the coordinates of the given k - point,
(index value = 2) an array containing all of the projections for that k - point.

```
Projections[dir0_] := Module[
   {
    dir = dir0,
    data,
    vals, start, end,
    numK, kVecs,
    numBands,
    numAtoms,
    bandLength,
    bandData,
    bandStart, bandEnd,
    energy, proj, add},


   (* read the PROCAR file into the list named "data": *)
   data = Import[dir <> "/PROCAR", "Table"];

   (* determine number of k-points, number of bands, and number of ions *)
   vals = StringJoin[Table[ToString[s], {s, data[[2]]}]];

   start = StringPosition[vals, "#ofk-points:"][[1, 2]] + 1;
   end = StringPosition[vals, "#ofbands:"][[1, 1]] - 1;
   numK = ToExpression[ StringTake[vals, {start, end}] ];
   Print["K-points: ", numK];
   kVecs = Table[{0, 0, 0}, {numK}];

   start = StringPosition[vals, "bands:"][[1, 2]] + 1;
   end = StringPosition[vals, "#ofions:"][[1, 1]] - 1;
   numBands = ToExpression[ StringTake[vals, {start, end}] ];

   start = StringPosition[vals, "#ofions:"][[1, 2]] + 1;
   end = StringLength[vals];
   numAtoms = ToExpression[ StringTake[vals, {start, end}] ];

   bandLength = 4 + 3 * numAtoms;

   Print["Number of bands: ", numBands];


   (* If there is only 1 atom in the system, the amount of blank-
    space changes by 1 line, so we account for this with the variable, add *)
   add = 1;
   If[numAtoms == 1, add = 0];

   (* parse the file for all of the band data and k-point data *)
   bandData = Table[
     bandStart =
      6 + (bandLength + 1 + add) (band - 1) + ((bandLength + 1 + add) (numBands) + 3) (k - 1);
     bandEnd = bandStart + bandLength;

     If[band == 1, kVecs[[k]] = data[[bandStart - 2, {4, 5, 6}]]];
     energy = data[[bandStart, 5]];

     proj = Partition[data[[Range[bandStart + numAtoms + 4 + add, bandEnd]]], 2];
     proj = Table[p[[1, i]] + i p[[2, i]], {p, proj}, {i, 2, Length[p[[2]]]}];

      {energy, proj}
     ,
      {k, numK},
      {band, numBands}
     ];

      (* insert the k-
     point coords for each kpoint in the band structure for convenience *)
   bandData = Table[{kVecs[[k]], bandData[[k]]}, {k, Length[bandData]}];

   Return[bandData];

  ];
```

■ Function: OrbitalOverlap

***result* = OrbitalOverlap[*orbitalProjections1, orbitalProjections2, indexes1, indexes2*]**
This function returns a matrix (a 2D array) of the overlaps of the wavefunctions from systems 1 and 2. Elements are of the form: $<\psi_1 \mid \psi_2>$ as introduced in *equation* 1 of the text.

Parameters
orbitalProjections1: array of projections associated with a single k-point from system 1 as returned from the *Projections* function (see the usage section II below for an example of how to get the desired array of projections from the result of the *Projections* function).
orbitalProjections1: array of projections from system 2.
indexes1: list of atom indexes from system 1 that remain unchanged upon substitution to system 2.
indexes2: list of atom indexes from system 2 that remain unchanged upon substitution to system 1.

Returns
result is a 2D array whose row indexes are the KS orbital indexes from system 1, and whose column indexes are the KS orbital indexes from system 2.

```
OrbitalOverlap[orbitalProjections01_,
   orbitalProjections02_, indexes01_, indexes02_] := Module[
   {orbitalProjections1 = orbitalProjections01, orbitalProjections2 =
     orbitalProjections02, indexes1 = indexes01, indexes2 = indexes02},
   Return[
    Table[
```

$$\sum_{a=1}^{\text{Length[indexes1]}} \left(\sum_{i=1}^{\text{Length}[\psi[[\text{indexes1}[[a]]]]]} \psi[[\text{indexes1}[[a]], i]]^{*} \; \phi[[\text{indexes2}[[a]], i]] \right)$$

```
     ,
     {ψ, orbitalProjections1[[All, 2]]},
     {ϕ, orbitalProjections2[[All, 2]]}
     ]
    ];
   ];
```

■ Function: MOMatrix

result **= MOMatrix[** ***overlap*** **]**
This function creates the MO matrix (2D array) of *equation 2* from the text.

Parameters
overlap: orbital overlap 2D array created by the *OrbitalOverlap* function above.

Returns
result is a 2D array whose row indexes are the KS orbital indexes from system 1, and whose column indexes are the KS orbital indexes from system 2 (where systems 1 and 2 were defined in the use of the *OrbitalOverlap* function).

```
MOMatrix[overlap0_] := Module[
   {
    overlap = overlap0, overlapSquared
   },

   overlapSquared = Table[
     Abs[overlap⟦i, j⟧]^2
     ,
     {i, Length[overlap]},
     {j, Length[overlap⟦1⟧]}
    ];
   Return[
    Table[
     overlapSquared⟦i, j⟧ /
      Max[{Max[overlapSquared⟦i, All⟧], Max[overlapSquared⟦All, j⟧]}]
     ,
     {i, Length[overlap]},
     {j, Length[overlap⟦1⟧]}
    ]
   ];
  ];
```

Function: cFunction

cFunction[***x, y, z*** **]**
This is an example color function used to determine the color of line segments in an MO diagram.

Parameters
x: x-coordinate (between 0 and 1) representing the energy (normalized to between 0 and 1) of the orbital at which a line segment either begins or ends.
y: y-coordinate. 0 if located at the bottom system (system 2), 1 if located at the top system (system 1).
z: magnitude of overlap between the orbital of system 1 and system 2 (normalized to be between 0 and 1).

Global Parameters
cutoff: the global variable cutoff is set externally since it is also used in the *MOSpectrum* below.

Returns
color based on x,y,z input.

```
cFunction = Function[
   {x, y, z},
   colorVal = Hue[x, (z - cutoff)/(1 - cutoff), 1];
   If[z == 0, colorVal = White];

   colorVal
  ];
```

Function: MOSpectrum

result **= MOSpectrum[** ***system1Projections, system2Projections, indexes1, indexes2, colorFunction*** **]**
This function creates the Graphics object used to display MO diagram connections between orbitals from

system1 and orbitals from system2, as seen in the MO diagrams from the text. Input is mostly the same as the *OrbitalOverlap* function defined above because this function calls the *OrbitalOverlap* function.

Parameters
system1Projections: array of projections associated with a single k-point from system 1 as returned from the *Projections* function (see the usage section II below for an example of how to get the desired array of projections from the result of the *Projections* function).
system2Projections: array of projections from system 2.
indexes1: list of atom indexes from system 1 that remain unchanged upon substitution to system 2.
indexes2: list of atom indexes from system 2 that remain unchanged upon substitution to system 1.
colorFunction: any standard *Mathematica* 3-variable color function, such as *cFunction* defined above.

Global Parameters
cutoff: the global variable cutoff is set externally (initialized below to .1) since it is generally also used in the *colorFunction*.

Returns
result is a *Mathematica* Graphics object containing line segments to be displayed as the connections between states of an MO diagram. While internally the energies are scaled to lie between 0 and 1, the return value *result* is rescaled to lie in the original energy range of the input systems.

```
cutoff = .2;
Options[MOSpectrum] = {eF1 → 0, eF2 → 0};
MOSpectrum[system1Projections0_, system2Projections0_,
   indexes01_, indexes02_, colorFunction0_, OptionsPattern[]] := Module[
   {
    system1Projections = system1Projections0,
    system2Projections = system2Projections0,
    indexes1 = indexes01, indexes2 = indexes02,
    overlap, moMatrix,
    eMax, eMin, Δe, e1, e2, x1, x2, val, map,
    colorFunction = colorFunction0, spec,
    eF1Shift, eF2Shift
   },

   eF1Shift = OptionValue[eF1];
   eF2Shift = OptionValue[eF2];

   overlap =
    OrbitalOverlap[system1Projections, system2Projections, indexes1, indexes2];
   moMatrix = MOMatrix[overlap];

   system1Projections⟦All, 1⟧ -= eF1Shift;
   system2Projections⟦All, 1⟧ -= eF2Shift;

   eMax = Max[ Join[system1Projections⟦All, 1⟧, system2Projections⟦All, 1⟧] ];
   eMin = Min[ Join[system1Projections⟦All, 1⟧, system2Projections⟦All, 1⟧] ];
   Δe = eMax - eMin;
   map = Table[
     e1 = system1Projections⟦e1Idx, 1⟧;
     e2 = system2Projections⟦e2Idx, 1⟧;
     x1 = (e1 - eMin)/Δe;
     x2 = (e2 - eMin)/Δe;
     val = {
       moMatrix⟦e1Idx, e2Idx⟧,
       Line[{{x1, 1}, {x2, 0}}, VertexColors → {colorFunction[x1, 1,
            moMatrix⟦e1Idx, e2Idx⟧], colorFunction[x2, 0, moMatrix⟦e1Idx, e2Idx⟧]}]
       };
     If[val⟦1⟧ < cutoff, val = {0, {}}];
     val
     ,
      {e1Idx, Length[system1Projections]},
      {e2Idx, Length[system2Projections]}
     ];
   map = Flatten[map, 1];
   map = Sort[map, #1⟦1⟧ < #2⟦1⟧ &];

   spec = Graphics[Flatten[map⟦All, 2⟧]];
   Return[ Graphics[
     Translate[Scale[spec⟦1⟧, {Δe, 1}, {0, 0}], {eMin, 0}]
     ] ];
   ];
```

II. Initialization (MO Energy Partitioning Functions)

The primary functions of interest are the *OrbitalEnergyContributions* function and the *ListWriteVASPBonds* function, defined first and second in this section, respectively. All other functions shown here support these functions and are either called directly from them or other supporting functions (*see section IV for usage below*).

■ Function: OrbitalEnergyContributions

***result* = OrbitalEnergyContributions[*dirSystems, dirVac, sysIndexes, vacIndexes, eigenValsRanges*]**
This function generates a list of energies, each energy corresponding to one of the vacuum system MOs already calculated in *dirVac*. These energies are the terms denoted as $\Delta E_{A\to B}^{e,\epsilon-\mathrm{eff}}$ in Eq. S4.

Parameters
dirSystems: array of directory names where each interpolated adsorption system has undergone SCF calculations as well as the force calculations of *ListWriteVASPBonds*. These interpolated systems also include the ground state system and the transition state system as the first and last element, respectively - and each interpolation is listed in order.
dirVac: directory name where the calculated vacuum system can be found.

sys*Indexes*: list of atom indexes from any of the adsorption systems (it is assumed that each adsorption system has the same set of indexes) that remain unchanged upon substitution into the vacuum system.
vac*Indexes*: list of atom indexes the vacuum system that remain unchanged upon substitution into any one of the adsorption systems.
eigenValsRanges: ordered pairs of energy ranges (slices) that forces have been calculated for. These pairs must be the same as were used for *ListWriteVASPBonds* below.

Returns
result is a list of energies corresponding to each of the vacuum system MOs.

```
OrbitalEnergyContributions[dirSystems0_, dirVac0_,
   sysIndexes0_, vacIndexes0_, eigenValsRanges0_] := Module[
   {
    dirSystems = dirSystems0, dirVac = dirVac0, eigenValsRanges = eigenValsRanges0,
    sysIndexes = sysIndexes0, vacIndexes = vacIndexes0,
    vacProjections, overlapSys, forceData, allDR, eigenValsMol, eigenValsSys,
    atoms1, atoms2, ϵList, αList, val, dEe, F1, F2, dR, ΔEe, ΔEeTotal
   },

   vacProjections = Projections[dirVac]⟦1, 2⟧;

   overlapSys = Table[
     OrbitalOverlap[ vacProjections,
      Projections[dirSystems⟦n⟧]⟦1, 2⟧, vacIndexes, sysIndexes ]
     ,
     {n, Length[dirSystems]}
    ];

   (* read in the calculated forces
    for each energy range of each adsorption system *)
   forceData = Table[
     GetBondData[d,
      Range[Length[eigenValsRanges]], EigenValueRanges → eigenValsRanges],
     {d, dirSystems}
    ];

   (* based on the atomic coordinates of each adsorption system
     (each being a system that is an interpolation of atomic coordinates
      between the ground state system and the transition state system),
   determine the displacement, dR, for each atom in moving from
    one interpolated position to the next *)
   allDR = Table[
     {atoms1, cell} = ReadCONTCAR[dirSystems⟦i⟧];
     {atoms2, cell2} = ReadCONTCAR[dirSystems⟦i + 1⟧];
     atoms2⟦All, {1, 2, 3}⟧ - atoms1⟦All, {1, 2, 3}⟧
     ,
     {i, Length[dirSystems] - 1}
    ];

   (* KS Eigenvalues of the molecule in vacuum *)
```

```
eigenValsMol = GetEigenValues[dirVac] - GetFermiEnergy[dirVac];

(* KS Eigenvalues for each interpolated adsorption
 system (each set of eigenvalues is an element of this list,
  one element per interpolated system) *)
eigenValsSys = Table[
  GetEigenValues[dir] - GetFermiEnergy[dir],
  {dir, dirSystems}
 ];

(* ϵList creates pairings between each consecutive set of interpolated
 systems (call a given pair sys1 and sys2, for example).  Then,
for each energy range (each slice of energies to be grouped together),
this list specifies which eigenvalue indexes of sys1 correspond to a given
 energy slice and which indexes of sys2 correspond to the same slice. *)
(* Note that the sizes of the energy slices are chosen by the input
 eigenValsRanges, and therefore can be set to be as small as needed -
 section IV shows an example of setting eigenValsRanges. *)
ϵList = Table[
  Select[
   Range[ Length[eigenValsSys[[c + add]]] ],
   eigenValsRanges[[E, 1]] ≤ eigenValsSys[[c + add, #]] < eigenValsRanges[[E, 2]] &
  ]
  ,
  {E, Length[eigenValsRanges]},
  {c, Length[dirSystems] - 1},
  {add, {0, 1}}
 ];

(* each range of KS eigenvalues in the adsorption system can be thought of as
  being derived from a set of precursor states in the vacuum system -
 so in this sense, a given adsorption system energy range belongs
 to a given precursor state with some weight α1 and to some other
 precursor state with weight α2, etc.  αList is the listing of
 those weights for each energy slice in each adsorption system. *)
αList = Table[
  If[Length[ϵList[[E, c, add + 1]]] > 0,
   val = Mean[ overlapSys[[c + add, e, ϵList[[E, c, add + 1]]]]^2 ],
   val = 0
  ];
  val
  ,
  {e, Length[eigenValsMol]},
  {E, Length[eigenValsRanges]},
  {c, Length[dirSystems] - 1},
  {add, {0, 1}}
 ];

(* numerically approximate the integral of Eq. S4 using the trapezoidal rule. *)
dEe = Table[
  F1 = forceData⟦c, E⟧⟦All, {8, 9, 10}⟧;
  F2 = forceData⟦c + 1, E⟧⟦All, {8, 9, 10}⟧;
  dR = allDR⟦c⟧;
```

$$\sum_{i=1}^{\text{Length[F1]}} \left(\frac{\alpha\text{List⟦e, E, c, 1⟧} * \text{F1⟦i⟧} + \alpha\text{List⟦e, E, c, 2⟧} * \text{F2⟦i⟧}}{2} . \text{dR⟦i⟧}\right)$$

```
      ,
      {e, Length[eigenValsMol]},
      {E, Length[eigenValsRanges]},
      {c, Length[dirSystems] - 1}
     ];

    ΔEe = Table[
      Total[dEe⟦e, All, c⟧],
      {e, Length[eigenValsMol]},
      {c, Length[dirSystems] - 1}
     ];

    ΔEeTotal = Re[Table[Total[ΔEe⟦x, All⟧], {x, Length[ΔEe]}] ];
    Return[ΔEeTotal];
   ];
```

Function: ListWriteVASPBonds

The following two variables, *scriptHeader* and *scriptHeader24*, define possible headers one might want to include in the script for calculating orbital forces, if running VASP in a PBS style environment (i.e. these would be part of the script submitted with a qsub command).

```
scriptHeader = "#!/bin/sh \n" <>
   "#PBS-l walltime=900:00:00 \n" <>
   "#PBS-l nodes=1:ppn=8:proc8 \n\n" <>
   "cd $PBS_O_WORKDIR \n\n\n";

scriptHeader24 = "#!/bin/sh \n" <>
   "#PBS-l walltime=900:00:00 \n" <>
   "#PBS-l nodes=1:ppn=24:proc24 \n\n" <>
   "cd $PBS_O_WORKDIR \n\n\n";
```

ListWriteVASPBonds[*dir, eigenValsRanges*]
This function sets up the calculation of the electrostatic forces due to the KS orbitals (actually small consecutive ranges of KS orbitals) of the system specified in the directory, *dir*. Note that the sizes of the energy slices are chosen by the input eigenValsRanges, and therefore can be set to be as small as needed - section IV shows an example of setting eigenValsRanges.

Parameters
dir: directory in which the VASP input files are specified for a given configuration, as well as results from an SCF run.
eigenValsRanges: ordered pairs of energy ranges (slices) that forces must be calculated for.

Results

After being run, *ListWriteVASPBonds* generates a set of subdirectories, one for each energy range specified by *eigenValsRanges* in which the forces will be calculated (via VASP). Additionally, a script is generated that will appropriately set up and run each VASP job in each of the subdirectories when executed.

```
Clear[ListWriteVASPBonds];
Options[ListWriteVASPBonds] =
  {ScriptHeader → scriptHeader, VScript → "mpirun -np 8 gvasp5",
   Occupations → {}, Extension → "", WriteCharge → False};
ListWriteVASPBonds[dir0_, listEnergy0_, OptionsPattern[]] := Module[
   {script, dirName, dir = dir0, listEnergy = listEnergy0, sysName = "Bond determination",
    vScript, occ, eF, atoms, cell, ext, eMin, writeCharge, header},

   writeCharge = OptionValue[WriteCharge];
   vScript = OptionValue[VScript];
   ext = OptionValue[Extension];
   occ = OptionValue[Occupations];
   If[Length[occ] == 0, occ = Table[0, {Length[listEnergy]}]];
   header = OptionValue[ScriptHeader];

   (* The given listEnergies are with respect to Ef=0,
   but the java code wants Ef at whatever VASP came up with *)
   eF = GetFermiEnergy[dir];
   listEnergy += eF;

   (* put a copy of the java code into
    the directory where the data to be processed is located *)
   CopyDirectory["waveBond", dir <> "/waveBond"];

   script = header;

   {atoms, cell} = ReadCONTCAR[dir];

   (* we want to know the forces on each ion due only to the nuclei *)
   eMin = GetEigenValues[dir][[1]] - 2;  (* so we will populate the wavecar for the
     bondZero directory with all 0 occupations ranging from the minimum eigenvalue -
    2 eV (eMin) to the fermi energy + 2 eV (eF + 2) *)
   script = script <>
     "cd " <> "bondZero\n" <>
     "cp ../WAVECAR WAVECAR\n" <>
     "cp ../POTCAR POTCAR\n" <>
     "cp ../CONTCAR POSCAR\n" <>
     "cp ../KPOINTS KPOINTS\n" <>
     "java -cp ../waveBond main.ProjectWave " <>
     ToString[eMin] <> " " <> ToString[eF + 2] <> " " <> ToString[0] <> "\n" <>
     "mv WAVECAR.out WAVECAR\n" <>
     "date | tee executionOutput \n" <>
     vScript <> " | tee -a executionOutput \n" <>
     "date | tee -a executionOutput \n" <>
     "rm WAVECAR \n" <>
     (* remove some of the larger output files to save space *)
     "rm CHG \n" <>
     "rm CHGCAR \n" <>
     "rm AEC* \n\n" <>
     "cd ../\n" <>
     "echo 'bondZero' | tee -a executionOutput \n\n";

      (* it would be preferable to include dipole corrections,
   however this is not possible (see the next comment) - otherwise we would set the
       dipole center as follows: dCenter=Inverse[cellᵀ].Mean[atoms⟦All,{1,2,3}⟧]; *)
   WriteINCAR[dir <> "/bondZero", sysName, UseVDW → True, UseDipoleCorrections → False ,
    UseDipole → False, (*DipoleGeometricCenter→dCenter,*)(* dipole corrections
     to the energy cause vasp to crash (segmentation fault!) if the occupations
     in the WAVECAR file have been adjusted and MaxElectronicSteps is set to 0,
    as is the case here *)MaxIonicSteps → 1, MaxElectronicSteps → 0,
    OUTCARDir → dir, GridPrecision → Accurate, ElectronEnergyCutoff → 400,
    AugmentationCutoff → 700, UseWaveFunction → True, WriteCharge → writeCharge];
      (* It is not clear to what extent, if any,
   ommiting dipole corrections will affect the results,
   given a WAVECAR file that was previously converged using dipole corrections. *)

   Do[
    dirName = dir <> "/bond" <> ext <> ToString[i];

    WriteINCAR[dirName, sysName, UseVDW → True, UseDipoleCorrections → False,
     UseDipole → False, MaxIonicSteps → 1, MaxElectronicSteps → 0,
     OUTCARDir → dir, GridPrecision → Accurate, ElectronEnergyCutoff → 400,
     AugmentationCutoff → 700, UseWaveFunction → True, WriteCharge → writeCharge];

    If[Length[listEnergy⟦i⟧] == 0,
     listEnergy⟦i⟧ = listEnergy⟦i⟧ {1, 1} + {- .0001, .0001}];

    script = script <>
      "cd " <> "bond" <> ext <> ToString[i] <> "\n" <>
      "cp ../WAVECAR WAVECAR\n" <>
      "cp ../POTCAR POTCAR\n" <>
      "cp ../CONTCAR POSCAR\n" <>
      "cp ../KPOINTS KPOINTS\n" <>
      "java -cp ../waveBond main.ProjectWave " <> ToString[listEnergy⟦i, 1⟧] <>
      " " <> ToString[listEnergy⟦i, 2⟧] <> " " <> ToString[1] <> "\n" <>
      "mv WAVECAR.out WAVECAR\n" <>
      "date | tee executionOutput \n" <>
      vScript <> " | tee -a executionOutput \n" <>
      "date | tee -a executionOutput \n" <>
      "rm WAVECAR \n" <>
      "rm CHG \n" <>
      "rm CHGCAR \n" <>
      "rm AEC* \n\n" <>
      "cd ../\n" <>
      "echo 'bond" <> ext <> ToString[i] <> "' | tee -a executionOutput \n\n";
    ,
     {i, Length[listEnergy]}
   ];
```

```
    Export[dir <> "/runBonds.sh", script, "Text"];

   ];
```

Function: WriteINCAR

***result* = WriteINCAR[*dir*, *name*]**
This function generates a VASP INCAR file in the directory specified by *dir*.

Parameters
dir: directory in which to place VASP input files.
name: readable description of the system to be calculated.

Selected Options (→ default values)
UseVDW → False: whether or not to use the vdW-DF functional.
MaxIonicSteps → 400: maximum number of steps that VASP is allowed to move atoms in search of a minimum energy geometric configuration.
MaxElectronicSteps → 200: maximum number of electronic (SCF) steps that VASP is allowed to take in search of a minimum electronic energy configuration.
OUTCARDir → ".": folder in which an OUTCAR from a previous run of this system can be found.
UseWaveFunction → False: whether or not to use the WAVECAR file from a previous run to choose the starting wavefunction.

```
CG = 2;
RMMDIIS = 1;
DFPT = 7;
Ions = 2;
VariableCell = 3;
Gaussian = 0;
TetrahedronMethod = -5;
Accurate = "ACCURATE";
OptB88 = 1;
RevPBE = 2;
OptB86b = 3;
Clear[WriteINCAR];
Options[WriteINCAR] = {
    UsePreviousRunPositions → True,
    ForceCutoff → .03,
    ElectronEnergyCutoff → 400,
    AugmentationCutoff → 700,
    MaxElectronicSteps → 200,
    MinElectronicSteps → {},
    MaxIonicSteps → 400,
    RelaxationMethod → CG,
    RelaxingObjects → Ions,
    WriteCharge → False,
    UseWaveFunction → False,
    WriteWaveFunction → True,
    CourseGrid → {},
    FineGrid → {},
    PreviousFiles → {},
    EnergyInterval → {},
    WriteDOS → False,
    OUTCARDir → ".",
    UseDipoleCorrections → False,
    UseDipole → False,
    DipoleGeometricCenter → {0, 0, 0},
    WriteLocalPotential → False,
    WriteLocalHartreePotential → False,
    Smearing → {},
    SmearingWidth → .2,
    GridPrecision → {},
    ElasticImages → 0,
    ProcessorsPerBand → 1,
    UseVDW → False,
    VDWExchange → OptB86b,
    UseDFTD → False,
    Symmetry → {},
    Bands → {}
   };
WriteINCAR[dir0_, name0_, OptionsPattern[]] := Module[
    {dir = dir0, name = name0, out, forceCutoff, ionEnergyCutoff,
     electronEnergyCutoff, augmentationCutoff, maxElectronicSteps, maxIonicSteps,
     relaxationMethod, relaxingObjects, writeCharge, useWaveFunction,
     writeWaveFunction, courseGrid, fineGrid, nbands, prevFiles, energyInterval,
     getDOS, outcarDir, useDipoleCorrections, useDipole, writeLocalPotential,
     smearing, smearingWidth, gridPrecision, elasticImages, procsPerBand, vdW,
     writeLocalHartreePotential, dCenter, sym, atoms, cell, numPoints, bands,
     minElectronicSteps, useDFTD, vdWExchange, usePreviousRunPositions},

    usePreviousRunPositions = OptionValue[UsePreviousRunPositions];
    bands = OptionValue[Bands];
    forceCutoff = OptionValue[ForceCutoff];
    electronEnergyCutoff = OptionValue[ElectronEnergyCutoff];
    augmentationCutoff = OptionValue[AugmentationCutoff];
    maxElectronicSteps = OptionValue[MaxElectronicSteps];
    minElectronicSteps = OptionValue[MinElectronicSteps];
    maxIonicSteps = OptionValue[MaxIonicSteps];
    relaxationMethod = OptionValue[RelaxationMethod];
    relaxingObjects = OptionValue[RelaxingObjects];
    writeCharge = OptionValue[WriteCharge];
    useWaveFunction = OptionValue[UseWaveFunction];
    writeWaveFunction = OptionValue[WriteWaveFunction];
    courseGrid = OptionValue[CourseGrid];
    fineGrid = OptionValue[FineGrid];
    prevFiles = OptionValue[PreviousFiles];
    energyInterval = OptionValue[EnergyInterval];
    getDOS = OptionValue[WriteDOS];
    outcarDir = OptionValue[OUTCARDir];
    useDipoleCorrections = OptionValue[UseDipoleCorrections];
    useDipole = OptionValue[UseDipole];
```

```
dCenter = OptionValue[DipoleGeometricCenter];
writeLocalPotential = OptionValue[WriteLocalPotential];
writeLocalHartreePotential = OptionValue[WriteLocalHartreePotential];
smearing = OptionValue[Smearing];
smearingWidth = OptionValue[SmearingWidth];
gridPrecision = OptionValue[GridPrecision];
elasticImages = OptionValue[ElasticImages];
procsPerBand = OptionValue[ProcessorsPerBand];
vdW = OptionValue[UseVDW];
useDFTD = OptionValue[UseDFTD];
sym = OptionValue[Symmetry];
vdWExchange = OptionValue[VDWExchange];

If[outcarDir == ".", outcarDir = dir];

If[(Length[energyInterval] == 2) ⋁ (Length[bands] > 0),
 useWaveFunction = True;
 writeCharge = True;
 writeWaveFunction = False;
];

If[useWaveFunction, nbands = GetNBands[outcarDir]];

(* if the specified directory does not exist, then create it*)
If[DirectoryQ[dir] == False,
 CreateDirectory[dir];
];
SetDirectory[dir];

If[StringQ[prevFiles],
 CopyFile["INCAR", "INCAR." <> prevFiles];
 CopyFile["OUTCAR", "OUTCAR." <> prevFiles];
 CopyFile["POSCAR", "POSCAR." <> prevFiles];
 CopyFile["CONTCAR", "CONTCAR." <> prevFiles];
 CopyFile["OSZICAR", "OSZICAR." <> prevFiles];
 If[usePreviousRunPositions,
  DeleteFile["POSCAR"];
  CopyFile["CONTCAR", "POSCAR"];
 ];
];

out = OpenWrite["INCAR"];

WriteString[out, "SYSTEM=" <> name <> "\n\n"];
If[useWaveFunction,
 WriteString[out, "ISTART=1\n"];
 WriteString[out, "ICHARG=0\n"];
 ,
 WriteString[out, "ISTART=0\n"];
];
If[maxIonicSteps > 0,
 WriteString[out, "EDIFFG=-" <> ToString[forceCutoff] <> "\n\n"]];

WriteString[out, "NPAR=" <> ToString[procsPerBand] <> "\n\n"];

WriteString[out, "ENCUT=" <> ToString[electronEnergyCutoff] <> "\n"];
WriteString[out, "ENAUG=" <> ToString[augmentationCutoff] <> "\n\n"];
WriteString[out, "LREAL=Auto\n"];

(* If using this function to generate an
 INCAR file for calculations before the ListWriteVASPBonds step
 (e.g. converging the wavefunction for each interp system),
then dipole corrections are available,
though this is no longer the case when ListWriteVASPBonds is run. *)
If[useDipoleCorrections,
 WriteString[out, "IDIPOL=3\n\n"];
];
If[useDipole,
 WriteString[out, "LDIPOL=.TRUE.\n"];

 WriteString[out, "DIPOL=" <> ToString[dCenter⟦1⟧] <> "   " <>
   ToString[dCenter⟦2⟧] <> "   " <> ToString[dCenter⟦3⟧] <> "\n\n"];
];

If[writeWaveFunction,
 WriteString[out, "LWAVE=.TRUE.\n"],
 WriteString[out, "LWAVE=.FALSE.\n"];
];

If[writeLocalPotential ∧ ¬ writeLocalHartreePotential,
 WriteString[out, "LVTOT=.TRUE.\n"]
];

If[writeLocalHartreePotential,
 WriteString[out, "LVHAR=.TRUE.\n"]
];

If[(Length[energyInterval] == 2) ∨ (Length[bands] > 0),
 WriteString[out, "LPARD=.TRUE.\n"]
];

If[writeCharge,
 WriteString[out, "LCHARG=.TRUE.\n"];
 If[Length[energyInterval] ≠ 2,
  WriteString[out, "LAECHG=.TRUE.\n\n"]
 ],
 WriteString[out, "LCHARG=.FALSE.\n\n"];
];

If[Length[energyInterval] == 2,
 WriteString[out, "EINT= " <>
   ToString[energyInterval⟦1⟧] <> " " <> ToString[energyInterval⟦2⟧] <> "\n"];
 WriteString[out, "NBMOD=-3\n\n"];
 ,
 If[Length[bands] > 0,
   WriteString[out,
     "IBAND= " <> StringJoin[Table[ToString[b] <> " ", {b, bands}]] <> "\n"];
  ];
];

If[courseGrid ≠ {},
 WriteString[out, "NGX=" <> ToString[courseGrid⟦1⟧] <> "\n"];
 WriteString[out, "NGY=" <> ToString[courseGrid⟦2⟧] <> "\n"];
 WriteString[out, "NGZ=" <> ToString[courseGrid⟦3⟧] <> "\n\n"];
];

If[fineGrid ≠ {},
 WriteString[out, "NGXF=" <> ToString[fineGrid⟦1⟧] <> "\n"];
 WriteString[out, "NGYF=" <> ToString[fineGrid⟦2⟧] <> "\n"];
 WriteString[out, "NGZF=" <> ToString[fineGrid⟦3⟧] <> "\n\n"];
];

If[getDOS,
 WriteString[out, "LORBIT=12\n\n"];
];

If[relaxationMethod == DFPT,
 WriteString[out, "LEPSILON= .TRUE.\n"];
 WriteString[out, "NWRITE= 3\n"];
 maxIonicSteps = 1;
];

WriteString[out, "NELM=" <> ToString[maxElectronicSteps] <> "\n"];
WriteString[out, "NSW=" <> ToString[maxIonicSteps] <> "\n"];
WriteString[out, "IBRION=" <> ToString[relaxationMethod] <> "\n\n"];
WriteString[out, "ISIF=" <> ToString[relaxingObjects] <> "\n"];
WriteString[out, "SIGMA=" <> ToString[smearingWidth] <> "\n"];

If[NumberQ[minElectronicSteps] == True,
 WriteString[out, "NELMIN=" <> ToString[minElectronicSteps] <> "\n"]];

If[NumberQ[sym] == True, WriteString[out, "ISYM=" <> ToString[sym] <> "\n"]];

If[ ListQ[smearing] == False,
 WriteString[out, "ISMEAR=" <> ToString[smearing] <> "\n"];
];
If[ ListQ[gridPrecision] == False,
 WriteString[out, "PREC=" <> ToString[gridPrecision] <> "\n"];
];

If[elasticImages > 0,
 WriteString[out, "IMAGES=" <> ToString[elasticImages] <> "\n"];
];
```

```
 If[vdW == True,
  WriteString[out, "\n"];
  WriteString[out, "LUSE_VDW= .TRUE. \n"];
  WriteString[out, "AGGAC=0.0000 \n"];
  If[vdWExchange == OptB88,
   WriteString[out, "GGA=BO \n"];
   WriteString[out, "PARAM1=0.1833333333 \n"];
   WriteString[out, "PARAM2=0.2200000000 \n\n"];
   ,
   If[vdWExchange == RevPBE,
     WriteString[out, "GGA=RE \n"];
     ,
     If[vdWExchange == OptB86b,
       WriteString[out, "GGA=MK \n"];
       WriteString[out, "PARAM1=0.1234 \n"];
       WriteString[out, "PARAM2=1.0000 \n\n"];
       ,
       Print["Unknown vdW Exchange chosen"];
      ];
    ];
  ];

  If[FileExistsQ["vdW/vdw_kernel.bindat"] == False,
   SetDirectory[NotebookDirectory[] ];
   CopyFile["vdW/vdw_kernel.bindat", dir <> "/vdw_kernel.bindat"];
   ,
   Print["Warning: vdw_kernal.bindat not found in folder \"vdW\", please download
       the vdw_kernal file for VASP and place it in the \"vdW\" folder."];
  ];
  ,
  If[useDFTD == True,
    WriteString[out, "\n"];
    WriteString[out, "LVDW= .TRUE. \n"];
   ];
 ];

 Close[out];
];
```

■ Function: GetNBands

***result* = GetNBands[*dir*]**
This function returns the number of bands used in a previous VASP calculation performed in the directory, *dir*.

Parameters
dir: directory in which VASP output files reside - specifically, where OUTCAR can be found.

Returns
result is the number of bands.

```
GetNBands[dir0_] := Module[
   {dir = dir0, data, val},
   data = ReadOUTCAR[dir];
   val = {};
   Do[
    If[StringQ[s] ∧ StringMatchQ[s, "NBANDS*"],
     val = data[[i]];
    ]
    ,
    {i, Length[data]},
    {s, data[[i]]}
   ];
   Last[val]
  ];
```

Function: ReadOUTCAR

***result* = ReadOUTCAR[*dir*]**
This function reads in the OUTCAR file located in directory *dir* and formats the data into a table.

Parameters
dir: folder where the OUTCAR file is stored.

Returns
result is the content of the OUTCAR file in table form.

```
Clear[ReadOUTCAR];
Options[ReadOUTCAR] = {FileName → "OUTCAR"};
ReadOUTCAR[name0_, OptionsPattern[]] := Module[
   {name = name0, fName},
   fName = OptionValue[FileName];
   Import[name <> "/" <> fName, "Table"]
  ];
```

Function: GetBondData

***result* = GetBondData[*dir*]**
This function reads in the forces from all of the files generated by the *ListWriteVASPBonds* function and its associated script. The forces are processed according to the description in the supplementary material in order to determine the effective forces that each orbital contribute to the total forces.

Parameters
dir: parent directory in which each of the "bond" sub-directories can be found (as generated by *ListWriteVASP-Bonds*).
nums: indexes of the "bond" subdirectories from which to extract force data (typically this would be all of the indexes).

Returns
result is a list for which each element is a set of atomic data. A given set of atomic data is a list whose elements are of the form {x,y,z,"C","T","T","T"Fx,Fy,Fz} where x,y,z are the atom coordinates, "C" is the atomic symbol (here Carbon, but could be any other such as "H", "O", etc., the next three slots are T/F for

whether the atom was allowed to move during ionic steps, and Fx,Fy,Fz are the three Cartesian components of the force vector that defines the force imparted on the given atom by a particular grouping of KS orbitals. Each element of result corresponds to the particular grouping of KS orbitals that *ListWriteVASPBonds* assigned to a given "bond" sub-folder.

Notes about this function: It is assumed that VASP has been run on each system to calculate the forces on each ion by reading in a WAVECAR file and performing 0 electronic (scf) steps and 1 ionic (the forces on atoms are calculated) steps. These resultant forces then comprise the force due to the ions + that due to the electrons. To get the electronic force only, then, the forces from the ions must be subtracted off. Finally, to get the "effective" forces, the ionic forces must be added back in, but scaled by the occupation fraction of a given range (see supplement).

```
Clear[GetBondData];
Options[GetBondData] = {Extension → "", EigenValueRanges → {}};
GetBondData[dir0_, nums0_, OptionsPattern[]] := Module[
   {dir = dir0, data, atoms, result = {}, nums = nums0, scale, i, extension, atomsZero,
    dataZero, eigenValueRanges, eigenValues, nElecTotal, nElec, f, effectiveForces},

   extension = OptionValue[Extension];
   eigenValueRanges = OptionValue[EigenValueRanges];

   (* bondZero should contain data from setting all occupations to zero so that
    the resultant forces are just those of the nuclei *)
   dataZero = ReadForceOUTCAR[dir <> "/bondZero"];
   atomsZero = dataZero⟦1⟧;

   If[Length[eigenValueRanges] == 0,
    (* if no ranges have been specified,
    then choose ranges that should have only one band per partition -
     this assumes no 2 eigenvalues are within .0001 of each other. *)
    eigenValueRanges = DeleteDuplicates[
      Table[e * {1, 1} + {-.0001, .0001}, {e, eigenValues}] ]
   ];

   (* determine occupation fractions, f, for each energy range *)
   eigenValues = GetEigenValues[dir] - GetFermiEnergy[dir];
   nElecTotal = NumElectronsOUTCAR[dir] / 2.;
   nElec =
    Table[Length[Select[eigenValues, r⟦1⟧ ≤ # ≤ r⟦2⟧ &]], {r, eigenValueRanges}];
   f = (nElec / nElecTotal);

   Print["Total number of bands in full range: ", nElecTotal];
   Print["Bands in each range: ", nElec];

   Do[
    data = ReadForceOUTCAR[dir <> "/bond" <> extension <> ToString[i]];

    atoms = data⟦1⟧;

    (* to get the force just from the electronic density,
    we subtract off the force due to the ions. *)
    (* the x,y,z coordinates of each force vector are at indexes 8,9,10  *)
    atoms⟦All, {8, 9, 10}⟧ -= atomsZero⟦All, {8, 9, 10}⟧;

    (* add the effective forces from the nuclei *)
    effectiveForces = atomsZero⟦All, {8, 9, 10}⟧ * f⟦i⟧;
    atoms⟦All, {8, 9, 10}⟧ += effectiveForces;

    result = Append[result, atoms];
    ,
    {i, nums}
   ];

   Return[result];
  ];
```

Function: GetEigenValues

***result* = GetEigenValues[*dir*]**
This function returns the list of EigenValues that VASP has calculated for the system in the directory *dir*.

Parameters
dir: the directory in which the EIGENVAL file can be found.

Returns
result is a list of each eigenvalue in eV.

```
GetEigenValues[dir0_] := Module[
   {dir = dir0},
   Select[Drop[Import[dir <> "/EIGENVAL", "Table"], 8], Length[#] == 2 &]⟦All, 2⟧
  ];
```

Function: GetFermiEnergy

***result* = GetFermiEnergy[*dir*]**
This function returns the calculated Fermi energy from a VASP calculation.

Parameters
dir: directory where OUTCAR can be found.

Returns
result the Fermi Energy in eV.

```
GetFermiEnergy[dir0_] := Module[{dir = dir0},
   Last[FermiEnergiesOUTCAR[dir]]⟦1⟧
  ];
```

Function: GetFermiEnergiesOUTCAR

***result* = GetFermiEnergiesOUTCAR[*dir*]**
This function returns the list of the Fermi energy at each step during a VASP calculation. The final Fermi energy is the correct one for the relaxed system.

Parameters
dir: directory where OUTCAR can be found.

Returns
result the list of Fermi energies in eV.

```
Clear[FermiEnergiesOUTCAR];
Options[FermiEnergiesOUTCAR] = {FileName → "OUTCAR", Data → {}};
FermiEnergiesOUTCAR[name0_, OptionsPattern[]] := Module[
   {name = name0, fileName, data, line, energies},

   fileName = OptionValue[FileName];
   data = OptionValue[Data];
   If[data == {},
    data = Import[name <> "/" <> fileName, "Table"];
   ];

   energies = {};
   Do[
    If[StringQ[s] ∧ StringMatchQ[s, "E-fermi"],
      energies = Append[ energies, Select[ Flatten[data⟦i⟧], StringQ[#] == False & ] ];
     ];
    ,
    {i, Length[data]},
    {s, data⟦i⟧}
   ];

   energies
  ];
```

■ Function: NumElectronsOUTCAR

***result* = NumElectronsOUTCAR[*dir*]**
This function returns the number of electrons used in the VASP calculation.

Parameters
dir: the directory in which OUTCAR can be found.

Returns
result the number of electrons used in the calculation.

```
Clear[NumElectronsOUTCAR];
Options[NumElectronsOUTCAR] = {FileName → "OUTCAR"};
NumElectronsOUTCAR[name0_, OptionsPattern[]] := Module[
   {name = name0, fileName, data, line, nums},

   fileName = OptionValue[FileName];

   data = Import[name <> "/" <> fileName, "Table"];

   nums = {};
   Do[
    If[StringQ[s] ⋀ StringMatchQ[s, "NELECT"],
      nums = Append[ nums, Select[ Flatten[data⟦i⟧], StringQ[#] == False & ] ];
     ];
    ,
    {i, Length[data]},
    {s, data⟦i⟧}
   ];

   Flatten[nums]⟦1⟧
  ];
```

▪ Function: ReadForceOUTCAR

result **= ReadForceOUTCAR[*dir*]**
This function reads all of the forces on each atom at each ionic step of a VASP calculation from the OUTCAR file.

Parameters
dir: the directory where OUTCAR can be found.

Returns
result is a list of force data for each ionic step of a VASP calculation. A single element of the list is a collection of each atom's data in the form {x,y,z,"C","T","T","T"Fx,Fy,Fz} where x,y,z are the atom coordinates, "C" is the atomic symbol (here Carbon, but could be any other such as "H", "O", etc., the next three slots are T/F for whether the atom was allowed to move during ionic steps, and Fx,Fy,Fz are the three Cartesian components of the force vector that defines the total force imparted on the given atom.

```
Clear[ReadForceOUTCAR];
Options[ReadForceOUTCAR] = {FileName → "OUTCAR"};
ReadForceOUTCAR[name0_, OptionsPattern[]] := Module[
   {name = name0, data, start, end,
    fName, posTable, types, names, nums, l, posData, super},

   fName = OptionValue[FileName];
   {posData, super} = ReadCONTCAR[name, FileName → "POSCAR"]; (* should be POSCAR *)

   data = Import[name <> "/" <> fName, "Table"];

   start = {};
   end = {};
   Do[
    If[StringQ[s] ⋀ StringMatchQ[s, "(eV/Angst)"], start = Append[start, i + 2]];
    If[StringQ[s] ⋀ StringMatchQ[s, "drift:"], end = Append[end, i - 2]];
    ,
    {i, Length[data]},
    {s, data⟦i⟧}
   ];

   If[Length[start] == 0, Return[{}]];

   If[end⟦1⟧ < start⟦1⟧, end = Drop[end, 1]];
   (* VASP 5.2 formatting is a little different than earlier versions *)

   posTable =
    Table[data⟦Range[start⟦i⟧, end⟦i⟧]⟧, {i, Min[Length[start], Length[end]]}];

   Do[
    posTable⟦i, j⟧ = Insert[posTable⟦i, j⟧, posData⟦j, 4⟧, 4];
    posTable⟦i, j⟧ = Insert[posTable⟦i, j⟧, posData⟦j, 5⟧, 5];
    posTable⟦i, j⟧ = Insert[posTable⟦i, j⟧, posData⟦j, 6⟧, 6];
    posTable⟦i, j⟧ = Insert[posTable⟦i, j⟧, posData⟦j, 7⟧, 7];
    ,
    {i, Length[posTable]},
    {j, Length[posTable⟦i⟧]}
   ];

   posTable

  ];
```

▪ Function: ReadCONTCAR

***result* = ReadCONTCAR[*dir*]**
This function reads in the geometric data from a CONTCAR or POSCAR file (including the atomic coordinates, as well as the super-cell defined as a 3x3 matrix).

Parameters
dir: the directory where the CONTCAR/POSCAR file can be found.

Returns
result is a list of 2 elements: {atoms, cell}. cell is the 3x3 supercell matrix, while atoms is a list whose elements are of the form: {x,y,z,"C","T","T","T"} where x,y,z are the atom coordinates, "C" is the atomic symbol (here Carbon, but could be any other such as "H", "O", etc., the next three slots are T/F for whether the atom was allowed to move during ionic steps.

```
Clear[ReadCONTCAR];
Options[ReadCONTCAR] = {FileName → "CONTCAR", Types → {}};
ReadCONTCAR[name0_, OptionsPattern[]] := Module[
   {name = name0, posData, scale, a, pos,
    cart, fName, types, nums, names, types0, l, char, add, val},

   fName = OptionValue[FileName];
   types0 = OptionValue[Types];

   posData = Import[name <> "/" <> fName, "Table"];
   scale = posData⟦2, 1⟧;
   a = posData⟦{3, 4, 5}⟧⟦All, {1, 2, 3}⟧;


   add = 0;
   If[StringQ[posData⟦6, 1⟧],
    nums = posData⟦7⟧;
    types0 = posData⟦6⟧;
    add = 1
    ,
    nums = posData⟦6⟧;
   ];

   l = Position[nums, "!"];
   If[Length[l] > 0,
    nums = nums⟦Range[l⟦1, 1⟧ - 1]⟧;
   ];

   pos = Select[Drop[posData, 7 + add],
      ((Length[#] == 6) ⋁ (Length[#] == 3)) ⋀ (¬ MemberQ[#, "!"]) &];
   pos = pos⟦Range[Total[nums]]⟧;
   pos = Table[
      val = p;
      If[Length[p] == 3, val = Join[val, {"T", "T", "T"}]];
      val,
      {p, pos}
    ];

   If[
    char = Characters[posData⟦8 + add, 1⟧][[1]];
    (char == "c") ⋁ (char == "C"),
    cart = Table[scale * p, {p, pos⟦All, {1, 2, 3}⟧}],
    cart = Table[scale * aᵀ.p, {p, pos[[All, {1, 2, 3}]]}];
   ];
   pos⟦All, {1, 2, 3}⟧ = cart;



   If[types0 == {},
    types = GetAtomTypesFromPOTCAR[name];
    names = Flatten[Table[Table[types⟦i⟧, {nums⟦i⟧}], {i, Length[nums]}]];
    ,
    names = Flatten[Table[Table[types0⟦i⟧, {nums⟦i⟧}], {i, Length[nums]}]];
   ];

   Return[{Table[Insert[pos⟦i⟧, names⟦i⟧, 4], {i, Length[pos]}], scale * a}];
  ];
```

▪ Function: GetAtomTypesFromPOTCAR

***result* = GetAtomTypesFromPOTCAR[*dir*]**
This function determines which atoms were used in a calculation based on the atoms defined in the POTCAR file.

Parameters
dir: the directory in which the POTCAR file can be found.

Returns
result is a list of atom names, e.g. {"C","H","O","Cu"} indicating which elements were used in a calculation.

```
GetAtomTypesFromPOTCAR[dir0_] := Module[
   {dir = dir0, data, types, start, end, fileName},

   fileName = dir <> "/POTCAR";

   data = Import[fileName];
   types = {};

   Do[
    If[StringQ[d] == True,
     If[StringFreeQ[d, "VRHFIN"] == False,
      start = StringPosition[d, "="]⟦1, 1⟧;
      end = StringPosition[d, ":"]⟦1, 1⟧;
      types = Append[types, StringTake[d, {start + 1, end - 1}]];
     ]
    ],
    {d, data[[All, 1]]}
   ];
   Return[types];
  ];
```

III. Example Usage (MO Diagram Functions)

- **The first example shows how to generate the line segments of an MO diagram:**

The folders relative to the working directory in which the PROCAR files for the systems A and AQ are (or should be) placed:

```
system1Dir = "vaspData/moDiagram/oAnth";
system2Dir = "vaspData/moDiagram/AQ";
```

The parsed projections will be in the form of an array whose 1st index is the k-point index and whose 2nd index references either:
(1) the coordinates of the given k-point, or
(2) an array containing all of the projections for that k-point.

This code assumes gamma-point only, so we want the first k-point (the only one), and the array of projections for that k-point, so the indexes should be : [[1, 2]]

```
system1Projections = Projections[system1Dir]⟦1, 2⟧;
system2Projections = Projections[system2Dir]⟦1, 2⟧;
```

```
K-points: 1

Number of bands: 46

K-points: 1

Number of bands: 51
```

The C and H atoms common to both species (the unsubstituted atoms) are indexed by the numbers 1 - 22 for both systems. These indexes correspond to the order in which they were specified in the POSCAR file (note that *Mathematica* starts counting at 1, rather than 0):

```
indexesAQ = Range[1, 22]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22}
```

```
indexesAnth = Range[1, 22];
```

In order to get energy levels to line up so that the Fermi energies of both system are at zero, we will need to extract the Fermi energies of each system using the GetFermiEnergy function defined in section II:

```
eF1Val = GetFermiEnergy[system1Dir]
```

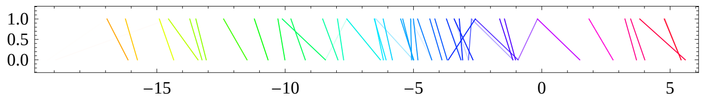
```
-4.604
```

```
eF2Val = GetFermiEnergy[system2Dir]
```

```
-5.6628
```

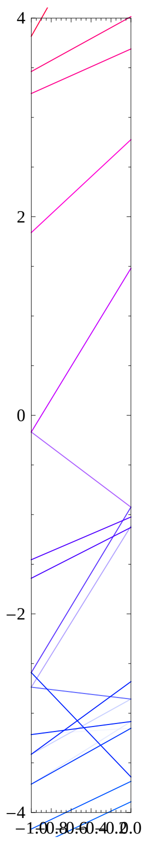
Now we can generate the desired MOSpectrum Graphics object using the results obtained so far:

```
g = MOSpectrum[system1Projections, system2Projections,
   indexesAnth, indexesAQ, cFunction, eF1 → eF1Val, eF2 → eF2Val];
Show[g, Frame → True, AspectRatio → .1]
```



For comparison to Fig. 2 from the main text (reproduced below), we can rotate 90°:

```
Show[
 Graphics[Rotate[g[[1]], 90 °, {0, 0}]],
 Frame → True, AspectRatio → 8, PlotRange → {{-1, 0}, {-4, 4}}
]
```



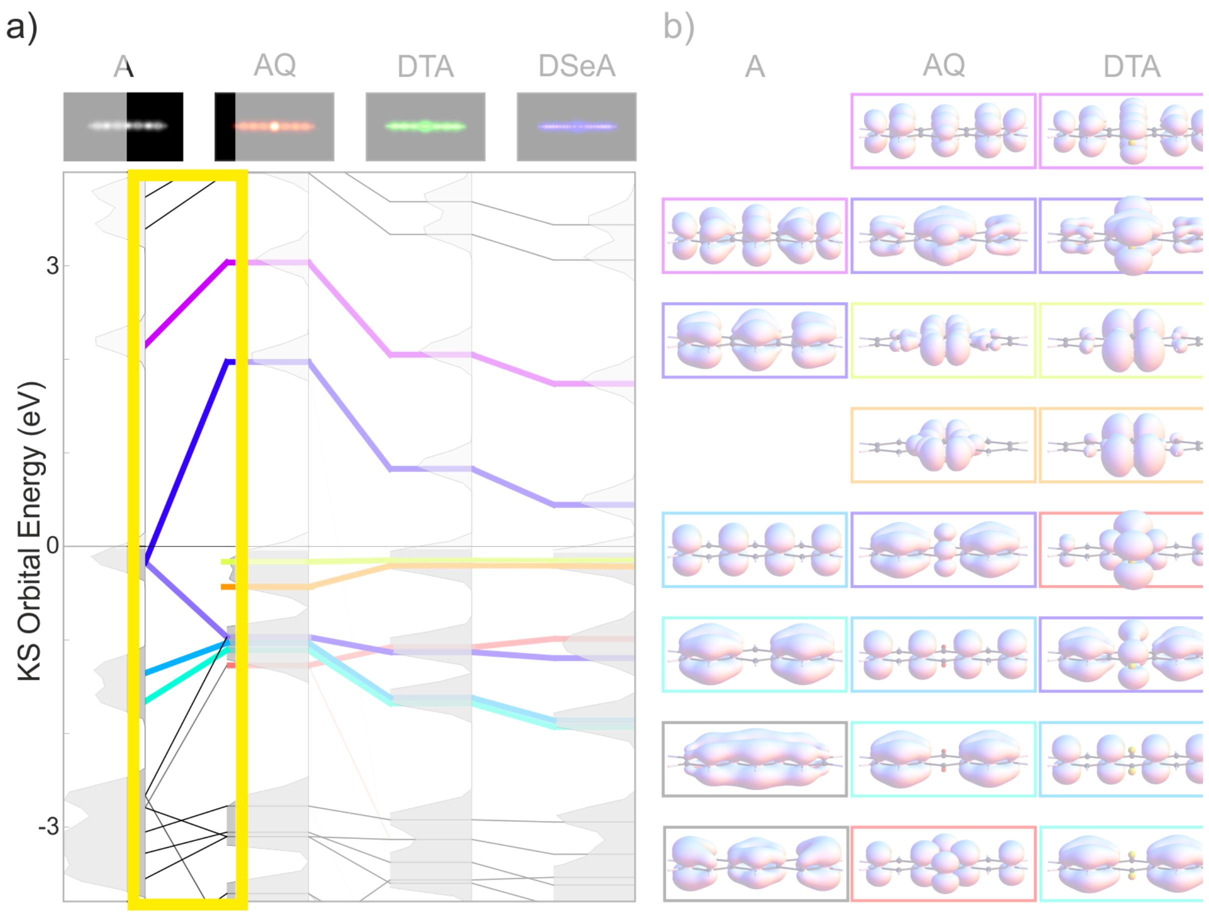


Fig. 2 from the main text. The links between states of A and those of AQ highlighted in the yellow box are reproduced by the code of this section.

- **The second example shows the workings of the *OrbitalOverlap* function and the *MOMatrix* function, both of which are used internally by the *MOSpectrum* function demonstrated above. This example requires the projections and indexes from above to be initialized.**

To minimize output, we only show 5x5 matrices for OrbitalOverlap and MOMatrix even though these are actually be 46x51 matrices.

```
overlap =
  OrbitalOverlap[system1Projections, system2Projections, indexesAnth, indexesAQ];
MatrixForm[overlap[[Range[5], Range[5]]]]
```

```
( 13.4089+0. i    -9.51153+0. i   29.6046+0. i    -0.423118+0. i   0.251316+0. i  )
( -0.076657+0. i  0.101887+0. i   0.5296+0. i     31.564+0. i      -0.027709+0. i )
( 6.06863+0. i    -4.43607+0. i   -5.97343+0. i   0.129853+0. i    24.9738+0. i   )
( 7.34758+0. i    10.4482+0. i    -0.113276+0. i  -0.056087+0. i   0.225384+0. i  )
( -0.032541+0. i  0.04444+0. i    0.061337+0. i   0.085326+0. i    0.016328+0. i  )
```

Note the "diagonal" nature of the MOMatrix below. Since it is not a square matrix, the "diagonal" does not begin at indexes 1,1 in this case but rather at indexes 1,3.

```
mo = MOMatrix[overlap];
MatrixForm[mo[[Range[5], Range[5]]]]
```

$$\begin{pmatrix} 0.205149 & 0.103225 & 1. & 0.000179696 & 0.0000720647 \\ 5.89818\times10^{-6} & 0.0000104196 & 0.000281521 & 1. & 7.70649\times10^{-7} \\ 0.0590489 & 0.031552 & 0.0407127 & 0.0000169246 & 1. \\ 0.107389 & 0.217148 & 0.0000146406 & 3.15747\times10^{-6} & 0.0000814473 \\ 1.80582\times10^{-6} & 3.3679\times10^{-6} & 4.29267\times10^{-6} & 7.30764\times10^{-6} & 4.27461\times10^{-7} \end{pmatrix}$$

At higher energies, there is a departure from diagonality (resulting from hybridization), particularly in the 4th column below (note the on-diagonal element of magnitude 0.7 and an off-diagonal element of magnitude 0.8 for example).

```
mo = MOMatrix[overlap];
MatrixForm[mo[[Range[30, 34], Range[33, 37]]]]
```

$$\begin{pmatrix} 0.000246144 & 0.000156222 & 0.0020451 & 0.832145 & 1.7067\times10^{-6} \\ 1. & 0.196598 & 0.000495225 & 0.000819615 & 0.000325338 \\ 1.75302\times10^{-6} & 0.00268792 & 1. & 0.00405599 & 0.00519209 \\ 0.0000475643 & 0.000717413 & 0.00127212 & 0.702769 & 1.2895\times10^{-7} \\ 0.0242762 & 0.149922 & 0.000240563 & 0.0000212039 & 2.23179\times10^{-7} \end{pmatrix}$$

IV. Example Usage (MO Energy Partitioning Functions)

- **The first example of this section shows how to set up the calculation of the forces imparted by each KS orbital energy range of the adsorption system:**

The following example assumes that the folders relative to the working directory in which the ground state structure (interp0) and the transition state structure (interp8) along with each intermediate interpolated structure (interp1 - interp7) have already been created and SCF calculations have been performed (with WAVECAR and PROCAR files already generated, as well as the typical VASP output files such as CONTCAR, EIGENVAL, and OUTCAR). *Note that in the following only interp0 and interp1 configurations are provided in the interest of limiting total file size - these are sufficient to see how the calculations work, but are not sufficient to reproduce the results of the study - one could however fill in the missing data by running VASP calculations for the transition state configuration and all interpolated states in between.* Also listed is the directory with the calculated results from the molecule in vacuum (the same as system2Dir in the example from section III).

```
dirSystems = {"vaspData/moDiagram/AQ/interp0", "vaspData/moDiagram/AQ/interp1"};
dirVac = "vaspData/moDiagram/AQ";
```

Ideally the forces due to each individual KS orbital for each adsorption system would be calculated; however, if resources are limited we can focus on narrow energy ranges of KS orbitals as specified here:

```
dE = 5 / 100.;
eigenValsRanges = Table[{e, e + dE}, {e, -5, 0, dE}]
```

```
{{-5., -4.95}, {-4.95, -4.9}, {-4.9, -4.85}, {-4.85, -4.8}, {-4.8, -4.75},
 {-4.75, -4.7}, {-4.7, -4.65}, {-4.65, -4.6}, {-4.6, -4.55}, {-4.55, -4.5},
 {-4.5, -4.45}, {-4.45, -4.4}, {-4.4, -4.35}, {-4.35, -4.3}, {-4.3, -4.25},
 {-4.25, -4.2}, {-4.2, -4.15}, {-4.15, -4.1}, {-4.1, -4.05}, {-4.05, -4.}, {-4., -3.95},
 {-3.95, -3.9}, {-3.9, -3.85}, {-3.85, -3.8}, {-3.8, -3.75}, {-3.75, -3.7},
 {-3.7, -3.65}, {-3.65, -3.6}, {-3.6, -3.55}, {-3.55, -3.5}, {-3.5, -3.45},
 {-3.45, -3.4}, {-3.4, -3.35}, {-3.35, -3.3}, {-3.3, -3.25}, {-3.25, -3.2},
 {-3.2, -3.15}, {-3.15, -3.1}, {-3.1, -3.05}, {-3.05, -3.}, {-3., -2.95},
 {-2.95, -2.9}, {-2.9, -2.85}, {-2.85, -2.8}, {-2.8, -2.75}, {-2.75, -2.7},
 {-2.7, -2.65}, {-2.65, -2.6}, {-2.6, -2.55}, {-2.55, -2.5}, {-2.5, -2.45},
 {-2.45, -2.4}, {-2.4, -2.35}, {-2.35, -2.3}, {-2.3, -2.25}, {-2.25, -2.2},
 {-2.2, -2.15}, {-2.15, -2.1}, {-2.1, -2.05}, {-2.05, -2.}, {-2., -1.95},
 {-1.95, -1.9}, {-1.9, -1.85}, {-1.85, -1.8}, {-1.8, -1.75}, {-1.75, -1.7},
 {-1.7, -1.65}, {-1.65, -1.6}, {-1.6, -1.55}, {-1.55, -1.5}, {-1.5, -1.45},
 {-1.45, -1.4}, {-1.4, -1.35}, {-1.35, -1.3}, {-1.3, -1.25}, {-1.25, -1.2},
 {-1.2, -1.15}, {-1.15, -1.1}, {-1.1, -1.05}, {-1.05, -1.}, {-1., -0.95},
 {-0.95, -0.9}, {-0.9, -0.85}, {-0.85, -0.8}, {-0.8, -0.75}, {-0.75, -0.7},
 {-0.7, -0.65}, {-0.65, -0.6}, {-0.6, -0.55}, {-0.55, -0.5}, {-0.5, -0.45},
 {-0.45, -0.4}, {-0.4, -0.35}, {-0.35, -0.3}, {-0.3, -0.25}, {-0.25, -0.2},
 {-0.2, -0.15}, {-0.15, -0.1}, {-0.1, -0.05}, {-0.05, 1.80411×10^-16}, {0., 0.05}}
```

The following is an example header for the script that will be generated in order to calculate the forces from each KS orbital. This is typical of a script that would be run in a Portable Batch System (PBS) environment (such as e.g. OpenPBS) - if vasp is being run on a cluster, it likely needs such a script:

```
Print[scriptHeader]
```

```
#!/bin/sh
#PBS-l walltime=900:00:00
#PBS-l nodes=1:ppn=8:proc8

cd $PBS_O_WORKDIR
```

This code loops through each interpolated adsorption system folder and generates the script as well as VASP input files necessary to calculate the forces for each energy interval specified in eigenValsRanges above. The option VScript specifies the actual command that should be called by the script in order to run VASP. If VASP is being run on a cluster in a PBS style system, the preferred command is likely something like the one shown below. On a single computer not running OpenPBS or the like, one would specify ScriptHeader->"" and VScript->"vasp" where "vasp" is the name of the VASP executable on the computer being run (gvasp5 in the example below). Finally note that if some of the files being generated already exist, *Mathematica* will complain about overwriting them, but these warnings can be ignored.

```
Do[
  ListWriteVASPBonds[dir, eigenValsRanges,
   ScriptHeader → scriptHeader, VScript → "mpirun -np 8 gvasp5"]
  ,
  {dir, dirSystems}
 ];
```

Note, since WAVECAR files have not been provided, there is no need to run the following on the example folders that have been provided - the results of such calculations are aleady provided as OUTCAR files. The runBonds.sh script requires the WAVECAR files to have already been generated for each interp folder from a previous VASP SCF run.
Having generated the scripts above, one would typically execute the generated script "runBonds.sh" in each of the interp folders specified above in dirSystems. For example, on a unix-like system using PBS one might type (where the ">" indicates a prompt and should not be typed):

```
> cd vaspData/moDiagram/AQ/interp0
> qsub runBonds.sh
```

or in the absence of PBS:

```
> cd vaspData/moDiagram/AQ/interp0
> bash runBonds.sh
```

One would do this for each of the adsorption system interpolations (interp0 through interp8, for example).

The runBonds.sh script found in a given interp folder generates a sub-folder for each of the energy ranges specified by eigenValsRanges, labeled bond1 through bondN, where N is the number of ranges specified. It then copies the WAVECAR file into each of those sub-folders, and modifies the occupations in each of the sub-folder WAVECAR files to only have the KS orbitals within the chosen energy range occupied (the WAVECAR file in bond1 would only have KS orbitals from the first energy range of eigenValsRanges occupied, for example). This is done by the script when it executes the ProjectWave java code (found in the waveBond folder). It also generates a sub-folder called bondZero in which all of the occupations have been set to zero (for determining the forces that each of the atoms with valence electrons removed imparts on each of the other ions).

It should be noted that this is a somewhat round-about way to get the forces associated with each KS orbital of a given adsorption system. This is because VASP does not directly output the forces due to each KS orbital so it is necessary to "turn all orbitals off except for those of interest" - an alternative approach would be to modify the VASP code (or one's favorite DFT code) directly to output this information.

- **The second example of this section shows how to calculate the barrier energy contribution from each vacuum system MO:**

Note that the variables: dirSystems, dirVac, and eigenValsRanges above must be defined (i.e. those *Mathematica* cells must be executed) for the following code to work.

Next, we specify the indexes of the C,H, and O atoms (as described in section III above, these correspond to the same order of atoms as specified in the POSCAR file) of both the vacuum system as well as the interpolated adsorption systems. In this case, for both the vacuum system and the adsorption systems, these indexes are the first 24:

```
vacIndexes = Range[24];
sysIndexes = Range[24];
```

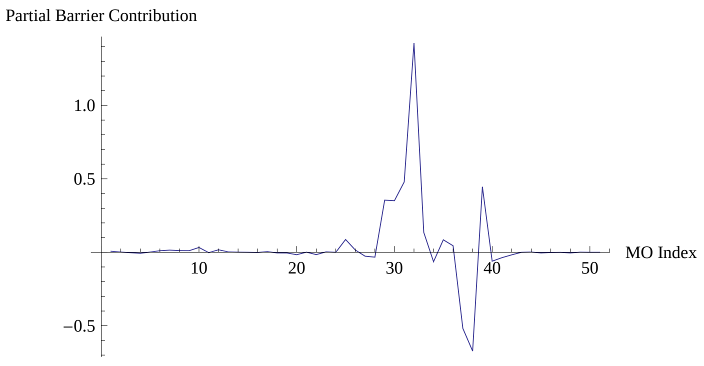
Now that everything is set up and all VASP calculations have been performed, we can calculate the energetic contribution from each of the vacuum system orbitals - be warned, this block of code can take a fairly long time to execute depending on how many interpolated systems are included:

```
ΔEeTotal = OrbitalEnergyContributions[
   dirSystems, dirVac, sysIndexes, vacIndexes, eigenValsRanges];
```

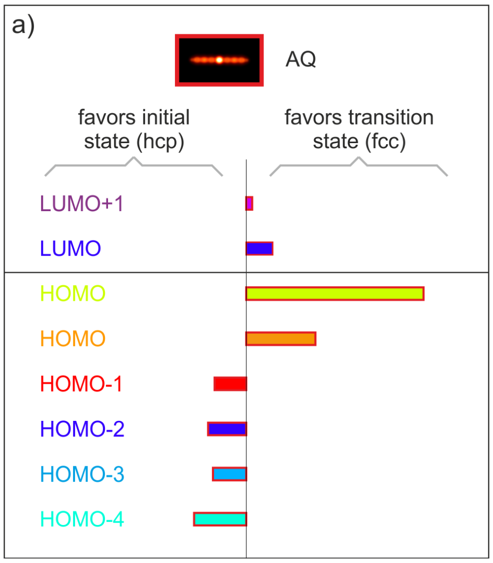
```
K-points: 1

Number of bands: 51

K-points: 1

Number of bands: 1629

K-points: 1

Number of bands: 1629

Total number of bands in full range: 1358.

Bands in each range:
 {5, 4, 4, 10, 5, 8, 10, 8, 11, 10, 4, 11, 8, 18, 12, 11, 16, 13, 12, 7, 12, 14, 9, 11, 14, 13,
  15, 12, 23, 8, 15, 20, 17, 18, 12, 18, 26, 22, 32, 26, 23, 17, 19, 20, 26, 23, 25, 18,
  15, 16, 21, 21, 15, 19, 25, 20, 22, 15, 19, 25, 24, 20, 25, 31, 30, 34, 37, 23, 11, 8,
  7, 6, 3, 4, 1, 3, 3, 1, 1, 3, 2, 2, 3, 1, 1, 1, 1, 2, 1, 0, 1, 2, 1, 3, 3, 1, 3, 4, 2, 1, 2}

Total number of bands in full range: 1358.

Bands in each range:
 {5, 3, 5, 10, 5, 8, 10, 8, 11, 10, 4, 11, 8, 18, 12, 11, 16, 13, 12, 7, 12, 14, 9, 11, 14, 13,
  15, 13, 22, 8, 15, 20, 17, 18, 12, 18, 26, 22, 31, 27, 24, 16, 19, 20, 26, 23, 25, 17,
  15, 17, 21, 21, 16, 18, 25, 20, 22, 15, 19, 25, 24, 20, 25, 31, 30, 34, 37, 23, 11, 8,
  7, 6, 3, 4, 1, 3, 3, 1, 1, 3, 2, 2, 3, 1, 1, 1, 1, 2, 1, 0, 1, 2, 1, 3, 3, 1, 3, 4, 2, 1, 2}
```

The energetic contribution to the barrier by each vacuum system MO can now be plotted (note that only 2 of the interpolated adsorption systems were used in this example, so this does not constitute the full barrier calculation):

```
ListPlot[ΔEeTotal, Joined → True, PlotRange → Full,
 AxesLabel → {"MO Index", "Partial Barrier Contribution"}]
```



In principle one could rotate the graph above by 90° and re-label the MO indexes to be HOMO, LUMO, etc. to reproduce the graph below (from Fig. 6a of the main text), however the graph above is incomplete since only interp0 and interp1 (out of 8) have been used.


a)
AQ
favors initial state (hcp)
favors transition state (fcc)
LUMO+1
LUMO
HOMO
HOMO
HOMO-1
HOMO-2
HOMO-3
HOMO-4