This solves for the intersection of y and z in terms of x and y.

```math
\[
\begin{align*}
\text{planesol} & = \text{Solve}\left[\{z == 1 + 2 x + 3 y, \ y == 3 - 2 z - 4 x\}, \{x, y\}\right]; \\
\text{planesol} & \;/\; \text{Simplify}
\end{align*}
\]
```

Here are the coordinates of any point on the line in terms of z.

```math
\[
\text{line} = \{x, y, z\} /. \text{planesol}[1] \;/\; \text{Simplify}
\]
```

Normalize to get the parallel unit vector.

```math
\[
\text{Normalize}[\{-7, 8, 10\}]
\]
```

Visual representation of this solution.
I2-2 (10 points)

A function that creates a matrix with the given specifications.

```math
randElementMat4x4 := RandomReal[{-1, 1}, {4, 4}]
```

This one creates a matrix based upon a 4D vector, magnitude 1 (using the Norm function).
```mathematica
In[167]:= randVectorMat4x4 := Module[
{rm},
    rm = randElementMat4x4;
    Table[rm[[i]] / Norm[rm[[i]]], {i, 1, Length[rm]}]
]

MatrixForm[randVectorMat4x4]
```

This verifies that the magnitude is 1 for each row.

```mathematica
In[169]:= test = randVectorMat4x4;
   Table[Norm[test[[i]]] == 1, {i, 1, Length[test]]]

Out[170]= {True, True, True, True}
```

**L2-3 (Handworked, 5 points)**

Creates and displays the matrices.

```mathematica
In[171]:= matrices =
{ {{a, c}, {c, b}},
  {{a, c + b Sqrt[-1]}, {c - b Sqrt[-1], 1}},
  {{a, -c}, {c, b}},
  {{a, c + b Sqrt[-1]}, {c + b Sqrt[-1], 1}}
};

Row@Table[MatrixForm[matrices[[i]]], {i, 1, 4}]
```

Displays the eigenvalues of each matrix assuming a, b, and c are real.
The first two matrices always have real eigenvalues.
\textbf{Table[FullSimplify[Eigenvalues[matrices[[i]]]], Assumptions -> \{a \in \text{Reals}, b \in \text{Reals}, c \in \text{Reals}\}], \{i, 4\}]

\begin{align*}
&\left\{\frac{1}{2} \left( a + b - \sqrt{(a - b)^2 + 4 c^2} \right), \frac{1}{2} \left( a + b + \sqrt{(a - b)^2 + 4 c^2} \right), \\
&\frac{1}{2} \left( 1 + a - \sqrt{1 + (-2 + a) a + 4 b^2 + 4 c^2} \right), \frac{1}{2} \left( 1 + a + \sqrt{1 + (-2 + a) a + 4 b^2 + 4 c^2} \right), \\
&\frac{1}{2} \left( a + b - \sqrt{(a - b)^2 - 4 c^2} \right), \frac{1}{2} \left( a + b + \sqrt{(a - b)^2 - 4 c^2} \right), \\
&\frac{1}{2} \left( 1 + a - \sqrt{1 + (-2 + a) a - 4 (b - i c)^2} \right), \frac{1}{2} \left( 1 + a + \sqrt{1 + (-2 + a) a - 4 (b - i c)^2} \right) \right\}
\end{align*}

Displays the eigenvectors of each matrix assuming a, b, and c are real.

\textbf{Row[Table[MatrixForm[FullSimplify[Eigenvectors[matrices[[i]]]], Assumptions -> \{a \in \text{Reals}, b \in \text{Reals}, c \in \text{Reals}\}], \{i, 4\}]]}

\begin{align*}
&\begin{pmatrix}
-a b + \sqrt{\frac{(a - b)^2 - 4 c^2}{2 c}} & 1 \\
-a b + \sqrt{\frac{(a - b)^2 + 4 c^2}{2 c}} & 1 \\
1 - i a - \sqrt{1 + (-2 + a) a + 4 b^2 + 4 c^2} & 1 \\
1 - i a + \sqrt{1 + (-2 + a) a + 4 b^2 + 4 c^2} & 1 \\
\end{pmatrix} \\
&\begin{pmatrix}
-a b + \sqrt{\frac{(a - b)^2 - 4 c^2}{2 c}} & 1 \\
-a b + \sqrt{\frac{(a - b)^2 + 4 c^2}{2 c}} & 1 \\
1 - i a - \sqrt{1 + (-2 + a) a - 4 (b - i c)^2} & 1 \\
1 - i a + \sqrt{1 + (-2 + a) a - 4 (b - i c)^2} & 1 \\
\end{pmatrix}
\end{align*}

\textbf{FullSimplify[Sqrt[1 - 2 a + a^2 + 4 c^2 + 4 b*I*I]]}

\begin{align*}
&\sqrt{1 + (-2 + a) a - 4 b + 4 c^2} \\
&\sqrt{1 + (-2 + a) a - 4 b + 4 c^2} \\
&\sqrt{1 + (-2 + a) a - 4 (b - i c)^2}
\end{align*}

\textbf{I2-4 (10 points)}

The given matrices.

\textbf{amat = Table[a[i, j], \{i, 1, 2\}, \{j, 1, 2\]]; bmat = Table[b[i, j], \{i, 1, 2\}, \{j, 1, 2\]]

The necessary condition is AB = BA or AB - BA = 0.
\texttt{commute = amat.bmat - bmat.amat}

\begin{verbatim}
\texttt{Out[178]=
\{-a\{2, 1\} b\{1, 2\} + a\{1, 2\} b\{2, 1\},
- a\{1, 2\} b\{1, 1\} + a\{1, 1\} b\{1, 2\} - a\{2, 2\} b\{1, 2\} + a\{1, 2\} b\{2, 2\}\},
\texttt{\{a\{2, 1\} b\{1, 1\} - a\{1, 1\} b\{2, 1\} + a\{2, 2\} b\{2, 1\} - a\{2, 1\} b\{2, 2\},
 a\{2, 1\} b\{1, 2\} - a\{1, 2\} b\{2, 1\}\})
\end{verbatim}

Finds the conditions such that $AB - BA = 0$.

\begin{verbatim}
\texttt{In[179]=
\texttt{eqs = Flatten[Table[commute[[i, j]] == 0, \{i, 2\}, \{j, 2\}], 1]}
\texttt{Out[179]=
\{-a\{2, 1\} b\{1, 2\} + a\{1, 2\} b\{2, 1\} == 0,
- a\{1, 2\} b\{1, 1\} + a\{1, 1\} b\{1, 2\} - a\{2, 2\} b\{1, 2\} + a\{1, 2\} b\{2, 2\} == 0,
a\{2, 1\} b\{1, 1\} - a\{1, 1\} b\{2, 1\} + a\{2, 2\} b\{2, 1\} - a\{2, 1\} b\{2, 2\} == 0,
a\{2, 1\} b\{1, 2\} - a\{1, 2\} b\{2, 1\} == 0\}}
\end{verbatim}

\begin{verbatim}
\texttt{In[180]=
\texttt{Solve[eqs, \{a\{1, 2\}, a\{2, 1\}\]}
\texttt{Out[180]=
\{(a\{1, 2\} \rightarrow \frac{- a\{1, 1\} b\{1, 2\} - a\{2, 1\} b\{2, 1\}}{b\{1, 1\} - b\{2, 2\}}, a\{2, 1\} \rightarrow \frac{- a\{1, 1\} b\{2, 1\} + a\{2, 2\} b\{2, 1\}}{b\{1, 1\} - b\{2, 2\}}\}}
\end{verbatim}

\section*{I2-5 (10 points)}

Start by constructing a vector that is a linear combination of a set of input vectors.

\texttt{createLinearDependent[listofVecs_] :=
\texttt{\texttt{Sum[RandomReal[\{-10, 10\}] listofVecs[[i]], \{i, 1, Length[listofVecs]\}]}}}

Test of the function.

\begin{verbatim}
\texttt{In[181]=
\texttt{createLinearDependent[\{\{1, 2, 3\}, \{4, 5, 6\}\]}}
\texttt{Out[181]=
\texttt{(36.2939, 48.8774, 61.461)}}
\end{verbatim}

Determinant $= 0$.

\begin{verbatim}
\texttt{In[183]=
\texttt{Det[Join[\{\{1, 2, 3\}, \{4, 5, 6\}\}], \{createLinearDependent[\{\{1, 2, 3\}, \{4, 5, 6\}\]}}
\texttt{Out[183]=
\texttt{0.}}
\end{verbatim}

Finds the nullity of the created matrix.
nullity4x4[nullity_] :=
Module[
    {linindset},
    linindset = RandomReal[{-1, 1}, {4 - nullity, 4}];
    Join[linindset,
        Table[createLinearDependent[linindset], {i, 1, nullity}]]
]

nullity4x4[2] // MatrixForm

As nullity increases, the matrix rank decreases.

4 - Table[MatrixRank[nullity4x4[i]], {i, 1, 3}]

The number of zeros in the set of eigenvalues increases as nullity increases.

countzeroes = Table[Count[evals[[i]], 0], {i, 1, 3}]

I2-6 (10 points; 2 points for rotation, 4 for reflection and 4 for inversion)

Here is the starter code.
`PolyhedronData["SnubCube"]`

Looking a GraphicsComplex in help browser will show that it takes two arguments. The first is a list of 3d points (which can be indexed as `point[[1]]`, `point[[2]]`...) and the second is a list of graphics objects that refer to the indexed points.

`FullForm[N@PolyhedronData["SnubCube"] // Short]`

```
Graphics3D[GraphicsComplex[List[\LeftSkeleton1\RightSkeleton],
Polygon[List[3, 1, 17], \LeftSkeleton36\RightSkeleton], List[8, 5, 7, 6]]]]
```

This uses a conditional pattern, it looks for instances of lists of three numbers and returns those cases. We only want the points here.
\[\text{In[192]}=\]
\[
\text{snubCubePoints} = \text{Cases[N@PolyhedronData["SnubCube"],}
\begin{align*}
\text{a_/} & \ (\text{NumberQ[a] \& \& ! IntegerQ[a]),} \\
\text{b_/} & \ (\text{NumberQ[b] \& \& ! IntegerQ[b]),} \\
\text{c_/} & \ (\text{NumberQ[c] \& \& ! IntegerQ[c])}, \text{Infinity}
\end{align*}
\]
\[\text{Out[192]}=\]
\[
\{-1.14261, -0.337754, -0.621226\}, \{-1.14261, 0.337754, 0.621226\}, \\
\{-1.14261, -0.621226, 0.337754\}, \{-1.14261, 0.621226, -0.337754\}, \\
\{1.14261, -0.337754, 0.621226\}, \{1.14261, 0.337754, -0.621226\}, \\
\{1.14261, -0.621226, -0.337754\}, \{1.14261, 0.621226, 0.337754\}, \\
\{-0.337754, -1.14261, 0.621226\}, \{-0.337754, 1.14261, -0.621226\}, \\
\{-0.337754, -0.621226, 1.14261\}, \{-0.337754, 0.621226, -1.14261\}, \\
\{0.337754, -1.14261, -0.621226\}, \{0.337754, 1.14261, 0.621226\}, \\
\{-0.621226, -1.14261, -0.337754\}, \{-0.621226, 1.14261, -0.337754\}, \\
\{-0.621226, -0.337754, 1.14261\}, \{-0.621226, 0.337754, -1.14261\}, \\
\{0.621226, -1.14261, 0.337754\}, \{0.621226, 1.14261, 0.337754\}, \\
\{0.621226, -0.337754, -1.14261\}, \{0.621226, 0.337754, -1.14261\}
\]

List of polygons. (The use of ___ refers to something repeated). 

\[\text{In[193]}=\]
\[
\text{snubCubePolygons} = \text{Cases[N@PolyhedronData["SnubCube"], List[a___Integer], Infinity}
\]
\[\text{Out[193]}=\]
\[
\{(3, 17), \{3, 17, 9\}, \{3, 19, 2\}, \{3, 9, 19\}, \{1, 4, 20\}, \{1, 20, 11\}, \{1, 11, 17\}, \\
\{2, 19, 12\}, \{2, 18, 4\}, \{2, 12, 18\}, \{4, 18, 10\}, \{4, 10, 20\}, \{17, 11, 13\}, \{19, 9, 15\}, \\
\{18, 12, 14\}, \{20, 10, 16\}, \{9, 21, 15\}, \{11, 23, 13\}, \{12, 24, 14\}, \{10, 22, 16\}, \\
\{13, 23, 7\}, \{13, 7, 21\}, \{15, 21, 5\}, \{15, 5, 24\}, \{16, 22, 6\}, \{16, 6, 23\}, \\
\{14, 24, 8\}, \{14, 8, 22\}, \{21, 7, 5\}, \{23, 6, 7\}, \{24, 5, 8\}, \{22, 8, 6\}, \{1, 3, 2, 4\}, \\
\{21, 9, 17, 13\}, \{24, 12, 19, 15\}, \{10, 18, 14, 22\}, \{11, 20, 16, 23\}, \{8, 5, 7, 6\}\}
\]

This gives the logical connection between points without regard to their coordinates, only where they are in list. 

\[\text{In[194]}=\]
\[
\text{snubCubeColoredPolygons} = \text{Table[}\{\text{FaceForm[Hue[RandomReal[]]],} \\
\text{Polygon[}\{\text{snubCubePolygons[[i]]}\}\}\}, \\
\{i, 1, \text{Length[snubCubePolygons]}\}\];
\]
\[\text{In[195]}=\]
\[
\text{snubGraphics[pts_] :=} \\
\text{Graphics3D[GraphicsComplex[pts, snubCubeColoredPolygons],} \\
\text{Lighting -> "Neutral", ViewPoint -> Front]}
\]

Visual representation of the polygons.
In[196]:= snubGraphics[snubCubePoints]


In[197]:= rotmat = RotationMatrix[Pi/3, {0, 1, 0}]

Out[197]= {\left\{\frac{1}{2}, 0, \frac{\sqrt{3}}{2}\right\}, \left\{0, 1, 0\right\}, \left\{-\frac{\sqrt{3}}{2}, 0, \frac{1}{2}\right\}}

Function to transform the coordinates of the points.

In[198]:= transformPoints[pts_, matrix_] := Table[matrix.pts[[i]], {i, 1, Length[pts]}]
In[199] = snubGraphics[transformPoints[snubCubePoints, rotmat]]

Out[199]=

Transforms given polygon.

In[200] = snubGraphics[
    transformPoints[snubCubePoints, {{-1, 0, 0}, {0, 1, 0}, {0, 0, 1}}]]

Out[200]=
An asymmetric object with six points

- This defines a set of colored polygons with indexed indices. The indices will refer to points that will be transformed as needed.

```math
AssymetricObject = \{ \{\text{Lighter[Blue]}, \text{Polygon}[\{1, 6, 2\}]\}, \\
\quad \{\text{Green}, \text{Polygon}[\{1, 2, 3, 4\}]\}, \{\text{Red}, \text{Polygon}[\{1, 4, 5, 6\}]\} \}
```

A graphical representation, given six points, this function creates an object ready for plotting

- Looking at the Help Browser for GraphicsComplex shows that this function takes points and then creates a graphics object associated with those points using the object created above.

```math
GraphicsAssymetricObject[pts_] := 
GraphicsComplex[pts, AssymetricObject]
```

A set of "generator" points

- These are the original points—they have the shape of a house, but then a pattern-replace method is used to displace and rotate each point.

```math
genPoints = 
\{\{0, 0\}, \{-0.5, 0.5\}, \{-0.5, -0.5\}, \{0, -0.5\}, \{0.5, -0.5\}, \{0.5, 0.5\}\}/. \\
\{\{a_, b_\} \rightarrow (0.1, 0.25) + 0.1 \text{RotationMatrix}[\pi/12].\{a, b\}\}
```

```math
genPoints[[1]]
```

\{0.1, 0.25\}, \{0.0387628, 0.285355\}, \{0.0646447, 0.188763\}, \\
\{0.112941, 0.201704\}, \{0.161237, 0.214645\}, \{0.135355, 0.311237\}\}
An example of an asymmetric object: this is kind of boring...one can do better...

Mapping Points back into unit cell: Modulus Mapping

1. This takes any point and finds it equivalent point in the primary unit square cell

\[
\text{SquareLatticeMod}[pt_] := \text{Mod}[pt, 1]
\]
Hexagonal lattice modulus mapping

- Working from the inside of the function calls out (i.e., the order of operation), the matrix \[
\begin{pmatrix}
1 & -1 \\sqrt{3}/2 \\
0 & 2 \\sqrt{3}/2
\end{pmatrix}
\] takes a vector in the hexagonal system and then maps it to a Cartesian frame. The modulus mapping function above can be applied directly in the Cartesian coordinate system. Finally the inverse of the first matrix is used to map the result back into the hexagonal system.

```math
\text{cartToHex} = \{\{1, 1/2\}, \{0, \sqrt{3}/2\}\};
```

```math
\text{hexToCart} = \text{Inverse}[\text{cartToHex}]
\{\{1, 1/\sqrt{3}\}, \{0, 2/\sqrt{3}\}\}
```

```math
\text{HexLatticeMod}[\text{pt_}] := (*take a pattern of numbers and use the modulus function to map back*)
\text{pt /.\{a_ /; \text{NumberQ}[a], b_ /; \text{NumberQ}[b]\} :>}
\text{cartToHex. Mod[hexToCart.\{a, b\}, 1]}
```

```math
\text{HexLatticeMod}[\{1.1, 0.99\}]
\{0.6, 0.123975\}
```

```math
\text{RandomReal}[\{0, 2\}, \{20, 2\}]
\{\{1.1815, 0.0501105\}, \{0.701729, 0.457317\}, \{0.433046, 1.93023\}, \{0.950424, 0.70391\},
\{1.35294, 0.583786\}, \{1.92407, 0.631786\}, \{1.85535, 1.32161\}, \{1.96879, 1.52197\},
\{1.73951, 1.55352\}, \{0.287979, 0.9988185\}, \{0.562582, 1.66186\}, \{0.337093, 1.17102\},
\{0.16801, 1.59379\}, \{0.00529066, 1.72497\}, \{1.06661, 1.90741\}, \{0.552723, 0.0126638\},
\{1.62724, 1.15432\}, \{0.0212442, 1.34539\}, \{1.63674, 1.97756\}, \{1.95014, 1.83545\}\}
```
An example of mapping back to the "fundamental hexagonal cell"

This shows that all points get mapped back—this is a demonstration that the function, at least, has proper behavior.

```
ListPlot[HexLatticeMod /@ RandomReal[{{0, 10}, {2000, 2}}],
PlotRange -> {{0, 2}, {0, 2}}, AspectRatio -> 1]
```

Rectangular lattice modulus mapping (set rectangle coordinates to (GoldenMean, 1))

Here is an example just like the hexagonal case above, but the mapping is a bit simpler.

```
rectToCart = {{1/GoldenRatio, 0}, {0, 1}};

cartToRect = Inverse[rectToCart]

{{GoldenRatio, 0}, {0, 1}}

RectLatticeMod[pt_] :=
  pt ./ ({a_ /; NumberQ[a], b_ /; NumberQ[b]} :> cartToRect . {rectToCart.{a, b}, 1})

RectLatticeMod[{1.1, 0.99}]

{1.1, 0.99}
```
ListPlot[RectLatticeMod @ RandomReal[{{0, 10}, {2000, 2}}],
PlotRange -> {{0, 2}, {0, 2}}, AspectRatio -> 1]

**Prism lattice modulus mapping (set prism coordinates to \(\sqrt{2}, 1\), Angle Pi/6)**

- This is a demonstration of the same method.

\[\text{toCart} = \{1 / \text{Sqrt}[2], \text{Sin}[\text{Pi} / 4] / \text{Sqrt}[2], 0, \text{Cos}[\text{Pi} / 4]\}\]

\[\{\left\{\frac{1}{\sqrt{2}}, \frac{1}{2}\right\}, \{0, \frac{1}{\sqrt{2}}\}\}\]

\[\text{Inverse[toCart]} \text{ // Simplify}\]

\[\{\sqrt{2}, -1\}, \{0, \sqrt{2}\}\]

\[\text{PrismLatticeMod}[pt_] :=\]

\[pt \/. \{a_ \rightarrow \text{NumberQ}[a], b_ \rightarrow \text{NumberQ}[b]\} \rightarrow\]

\[\{\left\{\frac{1}{\sqrt{2}}, \frac{1}{2}\right\}, \{0, \frac{1}{\sqrt{2}}\}\}.\text{Mod}[\{\{\sqrt{2}, -1\}, \{0, \sqrt{2}\}\}.\{a, b\}, 1]\]
Symmetry operations on a point

Define a function that translates points by a vector

- Using replacement rules again. We use chop to eliminate small numerical errors-- and their cumulative effect from many mappings.

\[
\text{TranslateSet}[\text{pointset}_\_, \text{transvect}_\_] := 
\text{Chop}[\text{pointset} /. 
\quad (\{a_\_/; \text{NumberQ}[a], b_\_/; \text{NumberQ}[b]\} \mapsto \{a, b\} + \text{transvect})]
\]

- For example:

\[
\text{TranslateSet}[\text{genPoints}, \{10, 100\}] \\
\{(10.1, 100.25), (10.0388, 100.285), (10.0646, 100.189), \\
(10.1129, 100.202), (10.1612, 100.215), (10.1354, 100.311)\}\]
Define a function that *rotates* points around origin by a counterclockwise angle

- This is a function that takes a set of points (which could be the vertices for several different instances of the asymmetric object) and rotates the counter-clockwise through an angle. The function takes an argument which is the name of a function that maps all the points back into the fundamental domain for the chosen symmetry.

The function creates the 2D rotation matrix from the input angle; uses a rule-replace to rotate each point in pointset and returns the results mapped back into the fundamental domain.

```math
RotateSet[pointset_, angle_, LatticeMap_] :=
Module[{rotmat},
  rotmat = RotationMatrix[angle];
  LatticeMap[
    Chop[pointset /. 
      ({a_/; NumberQ[a], b_/; NumberQ[b]} :> rotmat.{a, b})]
  ]
]
```

- For example:

```math
RotateSet[genPoints, Pi/4, SquareLatticeMod]
```

```
{{(0.893934, 0.247487), (0.925633, 0.229186), (0.912235, 0.179186),
  (0.937235, 0.222487), (0.962235, 0.265789), (0.875633, 0.315789)}}
```

Define a function that *mirrors* points through a plane defined by its direction and perpendicular offset from origin

- This symmetry operation works on the same principle. It takes the line across which pointset will be reflected and the distance (perpendicular bisector) of that line from the origin.

```math
MirrorSet[pointset_, line_, perpOffset_, LatticeMap_] :=
Module[{refmat, normalline, normvector},
  If[PossibleZeroQ[line[[2]]], normalline = {0, 1},
    normalline = {-1, line[[1]]/line[[2]]};
  normvector = Norm[normalline];
  refmat = ReflectionMatrix[normalline];
  LatticeMap[
    Chop[pointset /. 
      ({a_/; NumberQ[a], b_/; NumberQ[b]} :> refmat. 
        ({a, b} - perpOffset normvector) + perpOffset normvector)]
  ]
]
```
For example:

```
MirrorSet[genPoints, {-1, 1}, Sqrt[2], SquareLatticeMod]
```

```
{{(0.75, 0.9), (0.714645, 0.961237), (0.811237, 0.935355),
  (0.798296, 0.887059), (0.785355, 0.838763), (0.688763, 0.864645)}}
```

Define a function that *glide* points along a direction and then mirrors across that line.

This operates with the same basic method. It first mirrors and then translates in a direction specified by vector and a magnitude.

```
GlideSet[pointset_, vector_, mag_, perpOffset_, LatticeMap_] := Mod[
  TranslateSet[MirrorSet[pointset, vector, perpOffset, LatticeMap],
    mag vector / Norm[vector]],
    1
]
```

For example:

```
GlideSet[genPoints, {-1, 1}, Sqrt[2], 2]
```

```
GlideSet[{{(0.1, 0.25), (0.0387628, 0.285355), (0.0646447, 0.188763),
  (0.112941, 0.201704), (0.161237, 0.214645), (0.135355, 0.311237)}}, {-1, 1}, \sqrt{2}, 2]
```
Now, define a group of functions that add symmetry elements to the generator—thus building up point group

This is a function that takes all of the input points, translates them, and then creates a set of the original points plus their translations. Thus, it builds up the image by adding a particular operation. The function will report back if it creates any duplicate objects—these would be redundant symmetry operations.

```mathematica
setAddTranslation[pointset__, transvect__] := Module[{transset, psetchop, vecchop, joinset, unionset},
  psetchop = Chop[pointset];
  vecchop = Chop[transvect];
  transset = TranslateSet[psetchop, transvect];
  joinset = Join[psetchop, transset];
  (*Print[psetchop];
  Print[transset];*)
  unionset = Union[joinset];
  If[Length[joinset] =!= Length[unionset],
    Print["translation produced "] <>
    ToString[Length[joinset] - Length[unionset]] <>
    " redundant objects"];
  unionset
]
```

And a similarly for adding rotations incrementally.

```mathematica
setAddRotation[pointset__, angle__, LatticeMap__] := Module[{rotset, rotmat, psetchop, joinset, unionset},
  psetchop = Chop[pointset];
  rotmat = RotationMatrix[angle];
  rotset = RotateSet[psetchop, angle, LatticeMap];
  joinset = Chop[Join[psetchop, rotset]]; 
  unionset = Union[joinset];
  If[Length[joinset] =!= Length[unionset],
    Print["rotation produced "] <>
    ToString[Length[joinset] - Length[unionset]] <>
    " redundant objects"];
  unionset
]
```
And a similarly for adding mirrors incrementally.

```math
\text{setAddMirror}[\text{pointset}_-, \text{line}_-, \text{perpOffset}_-, \text{LatticeMap}_-] := \\
\text{Module}[\{\text{refset}, \text{psetchop}, \text{joinset}, \text{unionset}\}, \\
\text{psetchop} = \text{Chop}[\text{pointset}]; \\
\text{refset} = \text{MirrorSet}[\text{psetchop}, \text{line}, \text{perpOffset}, \text{LatticeMap}]; \\
\text{joinset} = \text{Chop}[\text{Join}[\text{psetchop}, \text{refset}]]; \\
\text{unionset} = \text{Union}[\text{joinset}]; \\
\text{If}[\text{Length}[\text{joinset}] \neq \text{Length}[\text{unionset}], \\
\text{Print}["reflection produced " <> \\
\text{ToString}[\text{Length}[\text{joinset}] - \text{Length}[\text{unionset}]] <> \\
" redundant objects"]; \\
\text{unionset} \\
]\]
```

And a similarly for adding glides incrementally.

```math
\text{setAddGlide}[\text{pointset}_-, \text{vector}_-, \text{mag}_-, \text{perpOffset}_-, \text{LatticeMap}_-] := \\
\text{Module}[\{\text{glideset}, \text{psetchop}, \text{joinset}, \text{unionset}\}, \\
\text{psetchop} = \text{Chop}[\text{pointset}]; \\
\text{glideset} = \\
\text{GlideSet}[\text{psetchop}, \text{vector}, \text{mag}, \text{perpOffset}, \text{LatticeMap}]; \\
\text{joinset} = \text{Chop}[\text{Join}[\text{psetchop}, \text{glideset}]]; \\
\text{unionset} = \text{Union}[\text{joinset}]; \\
\text{If}[\text{Length}[\text{joinset}] \neq \text{Length}[\text{unionset}], \\
\text{Print}["glide produced " <> \\
\text{ToString}[\text{Length}[\text{joinset}] - \text{Length}[\text{unionset}]] <> \\
" redundant objects"]; \\
\text{unionset} \\
]\]
```

The following functions use the previously defined rotations to create the 2-, 3-, 4-, and 6-fold rotation symmetries.

```math
\text{AddTwoFold}[\text{pointset}_-, \text{LatticeMap}_-] := \\
\text{setAddRotation}[\text{pointset}, \text{Pi}, \text{LatticeMap}]
```
AddThreeFold[pointset_, LatticeMap_] :=
Module[{psetchop, p1, p2, joinset, unionset},
  psetchop = Chop[pointset];
  p1 = RotateSet[psetchop, 2 Pi / 3, LatticeMap];
  p2 = RotateSet[psetchop, 4 Pi / 3, LatticeMap];
  joinset = Chop[Join[psetchop, p1, p2]];
  unionset = Union[joinset];
  If[Length[joinset] ≠ Length[unionset],
    Print["p3 produced "]
    ToString[Length[joinset] - Length[unionset]] <>
    " redundant object vertices"];
  unionset
]

AddFourFold[pointset_, LatticeMap_] :=
Module[{psetchop, p1, p2, p3, joinset, unionset},
  psetchop = Chop[pointset];
  p1 = RotateSet[psetchop, Pi / 2, LatticeMap];
  p2 = RotateSet[psetchop, Pi, LatticeMap];
  p3 = RotateSet[psetchop, 3 Pi / 2, LatticeMap];
  joinset = Chop[Join[psetchop, p1, p2, p3]];
  unionset = Union[joinset];
  If[Length[joinset] ≠ Length[unionset],
    Print["p4 produced "]
    ToString[Length[joinset] - Length[unionset]] <>
    " redundant object vertices"];
  unionset
]

AddSixFold[pointset_, LatticeMap_] :=
Module[{psetchop, p1, p2, p3, p4, p5, joinset, unionset},
  psetchop = Chop[pointset];
  p1 = RotateSet[psetchop, Pi / 3, LatticeMap];
  p2 = RotateSet[psetchop, 2 Pi / 3, LatticeMap];
  p3 = RotateSet[psetchop, Pi, LatticeMap];
  p4 = RotateSet[psetchop, 4 Pi / 3, LatticeMap];
  p5 = RotateSet[psetchop, 5 Pi / 3, LatticeMap];
  joinset = Chop[Join[psetchop, p1, p2, p3, p4, p5]];
  unionset = Union[joinset];
  If[Length[joinset] ≠ Length[unionset],
    Print["p6 produced "]
    ToString[Length[joinset] - Length[unionset]] <>
    " redundant object vertices"];
  unionset
]
Graphics functions

- This is somewhat inelegant. A graphics object representing the "grid" on which the plane-groups be drawn is defined by hand. It would be more clever to write functions for "mirror" "glide" objects, and "n-fold" graphics objects.

```
  {{0, 0}, {.5, 0}, {1, 0}, {1, .5}, {1, 1}, {.5, 1}, {0, 1}, {0, .5}},
  {{Thick, Dashed, Line[{{1, 5}}, Line[{{3, 7}}]},
  {Thick, Line[{{2, 4}}, Line[{{4, 6}}, Line[{{6, 8}}, Line[{{8, 2}}]}],
  {Thin, Line[{{1, 3}}, Line[{{3, 5}}, Line[{{5, 7}}, Line[{{7, 1}}]},
  {Line[{{1, 3}, {3, 5}, {5, 7}, {7, 1}}],
  Line[{{2, 6}}, Line[{{4, 8}}]]]]}
```

- The takes the points and draws all the objects using the previously defined "GraphicsAssymetricObject" in a list created by table. The second argument is the "background grid"

```
SetGraphics[pointset__, grid__] :=
  Show[Graphics[Table[
    GraphicsAssymmetricObject[pointset[[i]]],
    {i, 1, Length[pointset]}],
    ImageSize -> Medium, Axes -> True, AxesOrigin -> {0, 0},
    grid]
```
Example of "building p4 g m" (#12)

The asymmetric object:

```math
SetGraphics[genPoints, grid[12]]
```
SetGraphics[
  setAddMirror[genPoints, {1, -1}, Sqrt[2]/8, SquareLatticeMod],
  grid[12]
]
Add a glide

```
SetGraphics[
    setAddGlide[
        setAddMirror[genPoints, {1, -1}, Sqrt[2]/8, SquareLatticeMod],
        {1, 1}, Sqrt[2]/2, 0, SquareLatticeMod],
    grid[12]]
```
Add a p4, this reproduces the image in TSOM plane group #12

- The result is assigned to a symbol because it will be used below.

```math
pg[12] = SetGraphics[
  AddFourFold[
    setAddGlide[
      setAddMirror[genPoints, {1, -1}, Sqrt[2]/8, SquareLatticeMod], {1, 1}, Sqrt[2]/2, 0, SquareLatticeMod], SquareLatticeMod], grid[12]
]
```
Example of "building p3 1 m" (#15)

```math
{0, 0}, {1/2, 0},
{1, 0}, 1 + 1/4, Sin[Pi/3]/2}, {1 + 1/2, Sin[Pi/3]},
{1, Sin[Pi/3]}, {1/2, Sin[Pi/3]}, {1/4, Sin[Pi/3]/2}],

{Thick, Line[{1, 3}], Line[{3, 5}], Line[{5, 7}], Line[{7, 1}]},
{Thin, Line[{3, 7}]},
{Thick, Dashed, Line[{2, 6, 4, 8, 2}]}]
```

- Need to move the asymmetric object into the hexagonal primitive domain

```math
hexgenPoints = TranslateSet[genPoints, {0.1, -0.15}]
```

```math
{{0.2, 0.1}, {0.138763, 0.135355}, {0.164645, 0.0387628},
{0.212941, 0.0517037}, {0.261237, 0.0646447}, {0.235355, 0.161237}}
```
The asymmetric object:

SetGraphics[hexgenPoints, grid[15]]

Add a mirror

SetGraphics[
    setAddMirror[hexgenPoints, {1, 0}, 0, HexLatticeMod],
    grid[15]
]
Add a three fold to generate p3 1 m (#15)

```
  AddThreeFold[
    setAddMirror[hexgenPoints, {1, 0}, 0, HexLatticeMod],
    HexLatticeMod
  ],
  grid[15]
]
```

Here, creating a lattice by adding the lattice translation vectors is explored just for fun.

```
  setAddMirror[hexgenPoints, {1, 0}, 0, HexLatticeMod],
  HexLatticeMod
];
```

- The lattice translations.

```
hvectx = {1, 0};
hvecty = {1/2, Sqrt[3]/2};
```

- A set of indices representing some of the points in the lattice.

```
siteindices = Tuples[{0, 1, 2, 3}, 2]
```

{(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2),
 (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3)}
Using table, the lattice translations, and the indices to make a list of "lattice" points.

```math
hcoords = Table[siteindices[[1, 1]] hvectx + siteindices[[1, 2]] hvecty, 
    {i, Length[siteindices]}]
```

```
{(0, 0), \( \frac{1}{2}, \frac{\sqrt{3}}{2} \)}, 
(1, \sqrt{3}), \( \frac{3}{2}, \frac{3\sqrt{3}}{2} \)}, 
(1, 0), \( \frac{3}{2}, \frac{\sqrt{3}}{2} \)}, 
(2, \sqrt{3}), \( \frac{5}{2}, \frac{3\sqrt{3}}{2} \)}, 
(2, 0), \( \frac{5}{2}, \frac{\sqrt{3}}{2} \)}, 
(3, \sqrt{3}), \( \frac{7}{2}, \frac{3\sqrt{3}}{2} \)}, 
(3, 0), \( \frac{7}{2}, \frac{\sqrt{3}}{2} \)}, 
(4, \sqrt{3}), \( \frac{9}{2}, \frac{3\sqrt{3}}{2} \)}
```

- An example with four unit cells

```math
SetGraphics[Join[pg15, TranslateSet[pg15, \{1.5, Sqrt[3]/2\}],
    TranslateSet[pg15, \{.5, Sqrt[3]/2\}],
    TranslateSet[pg15, \{1, 0\}]], grid[15]]
```

- An example with all the indices defined above. The generated list had to be "Flattened" into a single list of points, where a point is a list of length 2.

```math
all = Flatten[Table[
    TranslateSet[pg15, hcoords[[i]]], {i, 1, Length[hcoords]}]], 1];
```
Building the remaining point groups:

Defining a map from cartesian to oblique and back...
grid[9] = Graphics[GraphicsComplex[Map[(cartToRect.# &) , {{0, 0}, {.5, 0}, 
{1, 0}, {1, .5}, {1, 1}, {.5, 1}, {0, 1}, {0, .5}, 
{1, .25}, (*9*) 
{1, .75}, (*10*) 
{0, .75}, (*11*) 
{0, .25}, (*12*) 
{.25, 0}, (*13*) 
{.75, 0}, (*14*) 
{.75, 1}, (*15*) 
{.25, 1} (*16*) 
}]], 
{Thick, Dashed, Line[{{12}, 9}, {10, 11}, {13, 16}, {14, 15}}]}, 
{Thick, Line[{{1}, 3}, {4}, 8}, {5, 7}, {2, 6}, {7, 1}, {3, 5}]}
]
pg[9] = SetGraphics[
  setAddMirror[
    AddTwoFold[
      setAddGlide[rectGenPoints, {1, 0}, .25, .75, RectLatticeMod],
      RectLatticeMod],
      {0, 1}, 0,
      RectLatticeMod],
    grid[9]
  ]
]
#12 p4gm (worked out above)

#13 p3

- The background grid. (I am going to use Map and a transform here. The methods defined above should work as well.
- As for hexgenPoints, need to move the asymmetric object into the oblique primitive domain

```math
    AddThreeFold[hexgenPoints, HexLatticeMod],
    grid[13]]
```
The background grid. (I am going to use Map and a transform here. The methods defined above should work as well.)

```
  GraphicsComplex[
    Map[(cartToHex.# &),
      {{0, 0}, {.5, 0}, {1, 0},
       {1, .5}, {1, 1}, {.5, 1}, {0, 1}, {0, .5},
       {.25, 0}, (*9*)
       {.75, 0}, (*10*)
       {1, .25}, (*11*)
       {1, .75}, (*12*)
       {.25, 1}, (*13*)
       {1, 1}, (*14*)
       {0, .25}, (*15*)
       {0, .75}(*16*)
    ]
  ],
  {Line[{{1, 3, 5, 7, 1}, {3, 7}}],
   {Thick, Line[{{1, 5}, {3, 6}, {2, 7}, {3, 8}, {4, 7}}]},
   {Thick, Dashed, Line[{{2, 4}, {4, 14},
      {12, 6}, {6, 8}, {8, 9}, {15, 2}, {10, 13}, {11, 16}}]}
  ]
]
```
- Need to move the points a little so as not to overlap

```
hexgenPointsAlt = TranslateSet[genPoints, {.2, -0.175}]
```

```
{{(0.3, 0.075), (0.238763, 0.110355), (0.264645, 0.0137628),
  (0.312941, 0.0267037), (0.361237, 0.0396447), (0.335355, 0.136237)}}
```

```
pg[14] = SetGraphics[
  AddThreeFold[
    setAddMirror[
      hexgenPointsAlt, cartToHex.{1, 1}, 0, HexLatticeMod],
      HexLatticeMod],
      grid[14]]
```
#15 p31m (worked out above)

#16 p6


- Need to move the points a little so as not to overlap

pg[16] = SetGraphics[
    AddSixFold[
        hexgenPointsAlt, HexLatticeMod],
    grid[16]]
The background grid. (I am going to use Map and a transform here. The methods defined above should work as well.)

```
  GraphicsComplex[
    Map[(cartToHex.# &),
      {{0, 0}, {0.5, 0}, {1, 0},
       {1, 0.5}, {1, 1}, {0.5, 1}, {0, 1}, {0, 0.5},
        {0.25, 0}, (*9*)
        {0.75, 0}, (*10*)
        {1, 0.25}, (*11*)
        {1, 0.75}, (*12*)
        {0.25, 1}, (*13*)
        {0.75, 1}, (*14*)
        {0, 0.25}, (*15*)
        {0, 0.75}(*16*)
    ]
  ],
  {Line[{{1, 3, 5, 7, 1}, {3, 7}}],
   Thick, Line[{{1, 5}, {2, 7}, {3, 6}, {3, 8}, {4, 7}}]},
   Thick, Dashed, Line[{{11, 16}, {10, 13}, {8, 9}, {6, 8}, {4, 14},
     {2, 4}, {15, 2}, {12, 6}, {2, 6}, {4, 8}, {4, 6}, {8, 2}}]
  ]
]
```
Need to move the points a little so as not to overlap

\[
\text{hexgenPointsAlt} = \text{TranslateSet}[\text{genPoints}, \{0.2, -0.175\}]
\]

\[
\{(0.3, 0.075), (0.238763, 0.110355), (0.264645, 0.0137628), \\
(0.312941, 0.0267037), (0.361237, 0.0396447), (0.335355, 0.136237)\}\]

\[
\text{pg[17]} = \text{SetGraphics[}
\text{AddSixFold[}
\text{setAddMirror[hexgenPointsAlt,}
\text{cartToHex.\{1, 1\}, 0, HexLatticeMod],}
\text{HexLatticeMod],}
\text{grid[17]}]
\]
Manipulate[pg[PlaneGroup],
{PlaneGroup, 1},
{1 -> "#1 p1 Oblique",
 2 -> "#2 p2 Oblique",
 3 -> "#3 pm Rectangular",
 4 -> "#4 pg Rectangular",
 5 -> "#5 cm Rectangular",
 6 -> "#6 p2mm Rectangular",
 7 -> "#7 p2mg Rectangular",
 8 -> "#8 p2gg Rectangular",
 9 -> "#9 c2mm Rectangular",
 10 -> "#10 p4 Square",
 11 -> "#11 p4mm Square",
 12 -> "#11 p4gm Square",
 13 -> "#13 p3 Hexagonal",
 14 -> "#14 p3m1 Hexagonal",
 15 -> "#15 p31m Hexagonal",
 16 -> "#16 p6 Hexagonal",
 17 -> "#17 p6mm Hexagonal"
}, ControlType -> PopupMenu,
SaveDefinitions -> True]
Final Graphical Result

Final compact graphical output

Group Problem G2-2

Start by creating some code to produce a radial distribution function from a set of given points

- Function to compute number in each annuli from a chosen origin, out to a given distance. This is not normalized by the total density. This is normalized by the particle radius--which is the natural dimension for normalization.

```plaintext
rdf[pointList_, radii_, bins_, origin_, rMax_] :=
Module[
  {distances, dr, areas, counts},
  dr = rMax / bins;
  distances =
    Table[Norm[pointList[[i]] - origin], {i, 1, Length[pointList]}];
  areas = Table[2 Pi (i dr) dr, {i, 1, bins}];
  counts = BinCounts[distances, {0, rMax, dr}];
  Table[{i dr / radii, counts[[i]] / areas[[i]]},
    {i, 1, Length[counts]}]
]
```
For example:

```
ListLinePlot[rdf[RandomReal[{-10, 10}, {100 000, 2}],
   1, 400, {0, 0}, 10], PlotRange -> All]
```

G-2-1

- First create a square lattice of points:

```math
squarelattice = Flatten[Table[i {1, 0} + j {0, 1}, {i, -20, 20}, {j, -20, 20}], 1];
Short[squarelattice]
v (-20, -20), (-20, -19), (-20, -18), (-20, -17), (-20, -16), (-20, -15), (-20, -14),
\((1667\ldots), (20, 14), (20, 15), (20, 16), (20, 17), (20, 18), (20, 19), (20, 20))
```

Using the origin: (there is a large peak at origin due to the "atom" sitting there). We use the hard packing radius as a measure of particle size.

```
GraphicsRow[
{lp = ListLinePlot[rdf[squarelattice, 1/2, 500, {0, 0}, 20],
   PlotRange -> All], Show[lp, PlotRange -> {{0, 10}, All}],
   Show[lp, PlotRange -> {{0, 10}, {0, 20}}]},
ImageSize -> Full
]
```
**G-2-2**

Using the \((1/2,1/2)\) : (this is more representative, the apparent curves are due to the "integer counts" in each bin)

\[
\text{GraphicsRow[}
\{lp = ListLinePlot[rdf[squarelattice, 1/2, 500, 0.5 {1, 1}, 20],
\quad \text{PlotRange \to All], Show[lp, PlotRange \to \{\{0, 10\}, All\}],}
\quad \text{Show[lp, PlotRange \to \{\{0, 10\}, \{0, 20\}\}}]\},
\quad \text{ImageSize \to Full}
\}\]

![Graphs showing apparent curves due to integer counts](image)

**G-2-(3&4)**

- We can reuse the transformation from G2-1, here we divide by the determinate to ensure that the areas are comparable.

\[
\text{cartToHex} = \left\{\left\{1, \frac{1}{2}\right\}, \left\{0, \frac{\sqrt{3}}{2}\right\}\right\} / (2 / \text{Sqrt}[3]);
\]

\[
\text{hexagonallattice} = \text{Table[}
\quad \text{cartToHex}.\text{squarelattice}[[i]], \{i, 1, \text{Length}[\text{squarelattice}]\}];
\]

\[
\text{ListPlot[hexagonallattice]}
\]

![Hexagonal lattice](image)
In this case, the distributions appear to be quite different depending on the origin. I believe it would make sense to pick many origins a random and compute an rdf to simulate what might be measured in an experiment.

```
GraphicsGrid[
{lp = ListLinePlot[rdf[hexagonallattice,
    Sqrt[3]/4, 500, {1, 1}, 15], PlotStyle -> {Blue, Thick},
    PlotLabel -> "Hexagonal at (1,1) ", PlotRange -> All],
    Show[lp, PlotRange -> {{0, 10}, All}]},
lp = ListLinePlot[rdf[hexagonallattice,
    Sqrt[3]/4, 500, .1 {1, 1}, 15], PlotStyle -> {Red, Thick},
    PlotLabel -> "Hexagonal at (.1,.1) " , PlotRange -> All],
    Show[lp, PlotRange -> {{0, 10}, All}]},
ImageSize -> Full
]
```

![Hexagonal at (1,1) graphs](image1)
![Hexagonal at (1,1) graphs](image2)
![Hexagonal at (1,1) graphs](image3)
First, consider random placement for any three disks, the probability that a disk can fit between them is quite low as the mean distance between the disks approaches a small number. In the algorithm that is described, the atoms are moved individually towards "empty space" and always leave a small gap between it and the nearest neighbor, the atoms expand until the specified density is achieved. As the distances between the neighbors becomes very small, the amount of available empty space limits the motion and thus the gaps. This reduction in gap size reduces the incremental expansion of each of the disks.

This is a repeat from last year; here are the results:

\[
\frac{\rho}{\rho_{\text{theo}}} = 0.33
\]

\[
\frac{\rho}{\rho_{\text{theo}}} = 0.5
\]
Compare the last one to an rdf for the hexagonal lattice above normalized to "roughly" the same sample size. The peaks in the high density amorphous plot begin to converge onto that of a characteristic crystal lattice—this might indicate the onset of six-fold nearest neighbors.

The code linked from the problem set works the following way. It takes a number of points and assigns a diameter to them. The points initially sit near the origin; so it is likely that many will overlap. The function picks a point at random and then "pushes" itself a short distance, but away from all overlapping disk centers. This creates more overlaps, but the outer disks tend to expand a bit. This process is to be repeated until there are no overlaps.

It turns out that this code has a flaw. When any particle sits at the centroid of overlapping neighbors, it won't move. This will have the effect of evolving to a fixed point that has one or more overlaps.

Here, I have rewritten the code to generate an amorphous structure using an algorithm with a similar "pushing" idea.

The following is a function that takes a list of points in two-dimensions (pnts) and makes one move to eliminate the overlaps (i.e., create a point-to-point separation of at least diam) from a chosen point (choice).

The algorithm operates by
1) Computing the displacements of the chosen point from all others (disps).

2) Using disps, finds any other disk of diam that overlaps. This method works the the following way: it strings together a bunch of "ands". The and returns false at the first occurrence of a false statement, thus it makes sense to put those which are less expensive to compute. It checks first to see if the comparisons are valid, and then eliminates all but the strip -diam < x < diam, then a similar strip in the y-direction; finally it computes the euclidian distance and eliminates all points within a radius of length diam. This last computation is the most expensive and so it is performed on as few points as possible.

3) Each overlapping particle is pushed away at least a distance diam (with a little randomness built-in). Those disks are likely to overlap their neighbors, and so this algorithm is repeated until all overlaps disappear. The algorithm is visualized below.

```mathematica
decompress[pts_, diam_, choice_] :=
  (*find an atom and push away its neighbors*)
Module[{cpt, numpts, intersecting, disps, tmpPts},
    numpts = Length[pts];
    cpt = pts[[choice]];
    (*Print[choice];*)
    disps = Map[ (# - cpt) &, pts];
    intersecting = Position[disps,
        a_ /; (Length[a] == 2 && Abs[a[[1]]] <= diam &&
            Abs[a[[2]]] <= diam && Norm[a] <= diam)];

    intersecting = DeleteCases[intersecting, {choice}];

    If[Length[intersecting] == 0, Return[pts]];
    (*Print[intersecting];*)
    tmpPts = pts;
    Table[
        (*Print[
            {i, tmpPts[[i]], cpt + diam disps[[i]]/Norm[disps[[i]]]}];*)
        tmpPts[[i]] = cpt + diam RandomReal[{1.0, 1.0001}, {2}]
        disps[[i]]/Norm[disps[[i]]],
        {i, Flatten[intersecting]}];
    tmpPts
]

(*This is the same algorithm as above, except a point is selected at random and passed as "choice" to the function above.*

decompress[pts_, diam_] :=
  decompress[pts, diam, RandomChoice[Range[Length[pts]]]]
```
This function displays the current state of all the points (data) as disks. Disccolors is a list of colors—if these are fixed then the evolution of the algorithm is easy to visualize. Additionally, text is drawn so that each particle in the list can be located by its position in data.

```math
\text{gob[data_, diam_, disccolors_] := Graphics[
\quad \text{GraphicsComplex[data, Table[[disccolors[[i]], Disk[i, diam/2],
                      \quad \{\text{Black, Text[ToString[i], i] \}}, \{i, Length[data]\}],
\quad \text{Axes \rightarrow True, ImageSize \rightarrow Large, PlotRange \rightarrow 4 \{\{-1, 1\}, \{-1, 1\}\},
\quad \text{PlotLabel \rightarrow "overlaps by " \rightarrow \text{ToString[minOverlap[data, diam]]}]
\quad (*draw a picture---here the code is modified
to compute the amount of overlap*)
\]}
```

Compute the minimum overlap. When this quantity goes to

```math
\text{minOverlap[data_, diam_] :=
\quad \text{Module[}
\quad \quad \{\text{nf, nearestNeighbor, neighbordists},
\quad \quad \text{nf} = \text{Nearest[data];
\quad \quad \text{nearestNeighbor} =
\quad \quad \quad \text{Table[Last[nf[data[[i]], 2]], \{i, 1, Length[data]\}];
\quad \quad \text{neighbordists} = \text{Min[Table[Norm[nearestNeighbor[[i]] - data[[i]]] -
\quad \quad \quad \quad \text{diam, \{i, 1, Length[data]\}]]}
\quad \]}
```

Start with a highly overlapping set of 400 disks and set their colors

```math
\text{randomdata} = \text{RandomReal[\{-1, 1\}, \{400, 2\}];
\text{diskColors} = \text{Table[Hue[RandomReal[]], \{Length[randomdata]\}]};
```

The graphical output is wrapped inside Dynamic so that the graphic updates whenever the "randomdata" is updated.

```math
\text{Dynamic[gob[randomdata, .2, diskColors]]}
```

Here is how it works by running once. Here a single particle is picked randomly and its neighbors are pushed away

```math
\text{randomdata} = \text{decompress[randomdata, .2]};
```

Here, the algorithm is run a bunch of times. It does 100 iterations (in Nest) before updating the graphic, and then it does this 1000 times. Running this twice is about enough to get a good data set to work with.

```math
\text{Do[randomdata} = \text{Nest[decompress[\#, .2] \&, randomdata, 100],
\quad \{i, 1, 1000\}];
```
Pick a radius for which we are going to compute the rdf, in effect the edge effects are being trimmed off.

```
estimatedRadius = 1.9
(*adjust this visually with following graphic, to remove any edge effects*)
```

1.9
Visualize the data set that will be used with the "cut-off" circle.

```
Show[gob[randomdata, .2, diskColors], Graphics[
  {Thick, Circle[Mean[randomdata], estimatedRadius]}], Axes -> True]
```

centerOfMass = Mean[randomdata]

\{0.00137753, 0.0332047\}
These are the "atoms" inside the cut-off circle that will be used in the rdf below.

```math
reducedDiskSetPositions = C =  \\
Cases[randomdata, a_ \rightarrow; \text{Norm}[a - \text{centerofMass}] \leq \text{estimatedRadius}]

\text{Length}[\text{reducedDiskSetPositions}]

291

reducedDiskSetIndices = \text{Position}[\text{randomdata},
  a_ \rightarrow; \text{Norm}[a - \text{centerofMass}] \leq \text{estimatedRadius}, 1]

\text{reducedDiskSetColors} = \text{Extract}[\text{diskColors}, \text{reducedDiskSetIndices}];
```
Show[gob[reducedDiskSetPositions, .2, reducedDiskSetColors],
Graphics[{{Thick, Circle[Mean[randomdata], estimatedRadius]}},
Axes -> True]
This shows that the algorithm achieves a density of about 80%. This is about 88% of the maximum theoretical. Not bad.

\[
\frac{\text{countableDisks \ Pi \ 1^2}}{\text{Pi \ estimatedRadius^2}}
\]

0.806094

\[
\text{ListLinePlot[rdf[reducedDiskSetPositions, 0.1, 400, centerofMass, estimatedRadius], PlotRange \rightarrow \{\{0, 10\}, \{0, 200\}\]}
\]

Compare this to a hexagonal rdf centered near the origin--the largest peaks seem to correlate (at least visually).

\[
\text{lp = ListLinePlot[rdf[hexagonallattice, Sqrt[3] / 4, 400, \{0.01, 0.01\}, estimatedRadius / (Sqrt[3] / 4)], PlotStyle \rightarrow \{\text{Blue, Thick}\}, PlotLabel \rightarrow "Hexagonal at (1,1)\", PlotRange \rightarrow \{\{0, 10\}, \{0, 50\}\]}
\]