Sample Section of *The Mathematica GuideBook for Graphics*

CHAPTER 2

2.4 Brillouin Zones of Cubic Lattices

Introduction

Similar to Section 1.6, in this last section of the 3D graphics chapter, we will deal with a larger example. We will have to implement a fair amount of functions to deal with this subject. We will on the way, as well at the final stages, make heavy use of visualizing the results and steps of the construction using 3D graphics. Similar to the mentioned equivalent section of Chapter 1 of the Programming volume [488✱] of the *GuideBooks*, we will make some use of *Mathematica*'s typesetting capabilities. All cells will be `InputForm` cells, but some special characters for `Part`, `Rule`, … will be used. Because efficiency is an important issue in the following section, often we will use constructions of the form *input* `// Timing` to better see the actual time needed for various operations.

Readers who are physicists, chemists, materials science majors and so on, will probably remembers the nice-looking pictures of polyhedra called Brillouin zones from the time the reader went to college/university. (In case the reader does not have one of these professions and/or have never heard the phrase Brillouin zone, this does not matter; just go on reading. I hope the reader will enjoy the pictures in the following anyway.) But the reader might have wondered why they never showed pictures of the, say, 11th Brillouin zone (which we would expect to look pretty funky taking into account how the first and second one look)—at least, the author always wondered about it. To cure this curiosity of the reader, in the following, we will explicitly calculate and visualize higher Brillouin zones for simple cubic, face-centered cubic and body-centered cubic lattices. (To refresh the memory about Brillouin zones, see [49✱], [26✱], [60✱], [258✱], [276✱], [262✱], [97✱], [539✱], [301✱], [495✱], [338✱], and [457✱]; for an elementary geometrical introduction, see [196✱].)

Now, let us become more technical. What is a Brillouin zone? Let us give a recursive definition.

The first Brillouin zone of a lattice Λ with lattice points $\vec{g}_i$ is the closure of the set of all points $\vec{x}$ such that $\left|\vec{x}-\vec{0}\right| \leq \left|\vec{x}-\vec{g}_i\right|$ for all $\vec{g}_i \neq \vec{0}$. (In other words, the first Brillouin zone is the Voronoi cell of the lattice around the origin.) The second Brillouin zone is formed by all points $\vec{x}$ such that $|\vec{x}-\vec{0}| \leq |\vec{x}-\vec{g}_i|$ for all $\vec{g}_i$ not already used in the inequalities of the first Brillouin zone. The third Brillouin zone is formed by all points $\vec{x}$ such that $|\vec{x}-\vec{0}| \leq |\vec{x}-\vec{g}_i|$ for all $\vec{g}_i$ not already used in the inequalities of the first and second Brillouin zone, the fourth is and so on. In other words, the $(n+1)$th Brillouin zones is the set of points that a line to them crosses exactly n bisector planes. (In computational geometry, a nth-order Brillouin zones would be called an nth-degree Voronoi region [132✱], [31✱] or the nth nearest point Voronoi diagram [364✱].)

This definition immediately suggests the following constructive algorithm for building Brillouin zones.

Fix one lattice point, and call it the origin. Construct line segments joining the origin with all lattice points. (When really doing the construction, we will, of course, only use a finite set of lattice points around the origin.) Erect perpendicular bisectors (in a 2D lattice, these are lines; in a 3D lattice, these are planes). These bisectors intersect each other typically quite often. We split the bisectors into pieces formed by intersections with the other bisectors. Now, take out all (finite) parts of the bisectors that bound the region that encloses the origin—this is the (boundary of the) first Brillouin zone. Now, again take out all parts of the bisectors that bound the region that encloses the origin —this is the (boundary of the) second Brillouin zone (the inner boundary of the second Brillouin zone is the boundary of the first Brillouin zone). Now, again take out all parts, and so on.

This section implements the construction of 3D Brillouin zones of cubic lattices. We will construct the first 20 Brillouin zones and will visualize them. The construction is fairly general, and by taking into account more lattice points, it is straightforward to construct the first 50 Brillouin zones (which of course needs more memory and more time).

```
In[1]:= (* to save memory *) $HistoryLength = 0;
```

We will split the construction of the individual regions into the following steps:

1. Construction and tessellation of the bisector planes: We will start with the lattice points near the origin inside a sphere of given diameter. Then, we will construct planes perpendicular to the lines from the origin to the lattice point at half the distance origin-lattice point. For each of these planes, we will calculate the intersection lines with all other planes. We will use the symmetry of the underlying lattice to calculate only “different” plane arrangements—for a cubic lattice, this reduces most of the time-consuming calculations by a factor of 48. Then, we will calculate the convex polygons inside each of the planes formed by the other planes.

2. Construction of the polygons in the symmetry unit: We will calculate all polygons from all bisector planes that fall into the symmetry unit.

3. Build the Brillouin zones from the polygons: Recursively, we will form the polytopes around the origin whose faces are the polygons from step 2 and are nearest to the origin.

4. Visualize the Brillouin zones: We will generate the polygons in the other 47 units and remove artificial polygon edges induced by the bounding planes of the symmetry unit.

5. Construction of Brillouin zones of a body-centered lattice.

6. Construction of Brillouin zones of a face-centered lattice.

In all of the implementation, we must care about efficiency, or we will not be able to generate the higher order zones. We will carry out all arithmetic using rational numbers, which avoids any rounding problems and identification problems of points with high degeneracies (typically, it is advantageous to use floating point numbers in graphics-related applications; here we have a counter example to this general rule). Other lattices types, hexagonal, for instance, are treatable in a similar way, but because the coordinates of the intersections of planes will contain radicals, `RootReduce` (see Chapter 1 of the Symbolics volume [490✱] of the *GuideBooks*) is needed for canonicalization. This would make things more time-consuming. (Another possibility would be to use high-precision arithmetic. This would require certain changes in the following code.)

In parts 1 to 4 of the construction, we will implement, test and explain all functions needed such that parts 5 and 6 become short, simple calls to the already-implemented functions.

1. Construction and Tessellation of the Bisector Planes

The function `latticePointListSC` generates all lattice points that we want to take into account. It returns all lattice points inside a sphere of radius δ around the origin. We start with a simple cubic lattice. (At the end, we will treat a body-centered and a face-centered lattice too.)

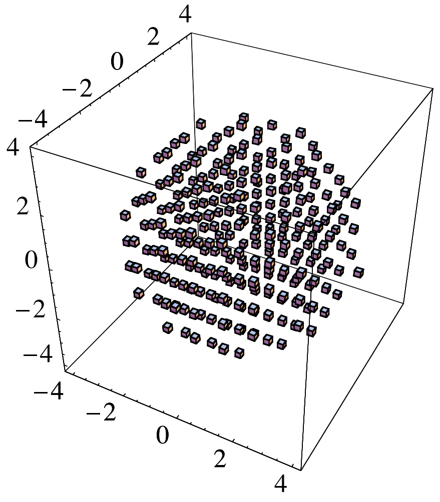
```
In[2]:= latticePointListSC[δ_] :=
          With[{n = Ceiling[δ]},
            Select[DeleteCases[
              Flatten[Table[{i, j, k}, {i, -n, n}, {j, -n, n}, {k, -n, n}], 2],
               (* without origin *){0, 0, 0}], #. # <= δ^2&]]
```

Taking into account all lattice points yields 256 lattice points inside a sphere of radius 4.

```
In[3]:= maxDist = 4;
        Length[latticePoints = latticePointListSC[maxDist]]

Out[4]= 256
```

Here are these lattice points.

```
In[5]:= Show[Graphics3D[Cuboid[# + 0.1 {1, 1, 1},
                               # - 0.1 {1, 1, 1}]& /@ latticePoints],
             Axes → True];
```



In the following calculations, we often have to normalize vectors to unit length. The pure function *n* will do this.

```
In[6]:= (* a function for normalizing vectors *)
        n = #/Sqrt[#.#]&;
```

Next, we construct the perpendicular planes in the middle of the lines *origin–latticePoint*. We denote a plane in the form `Plane[`*onePointOfThePlane*`,`*listOfTwoOrthogonalDirections*`]`. The function `toPlane` generates a plane (head `Plane`) of the bisector plane formed by the point *latticePoint*.

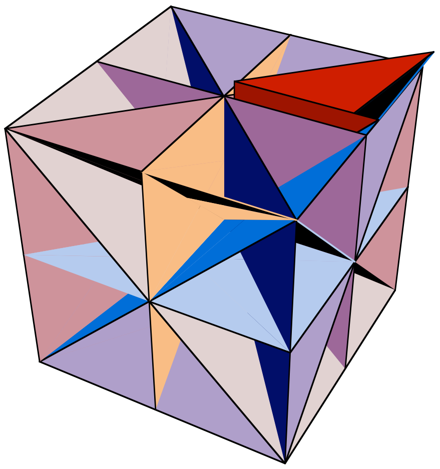
```
In[8]:= toPlane[p:latticePoint_] :=
        Plane[latticePoint/2,
            Which[(* lattice point is on a coordinate axis *)
                  p⟦1⟧ == 0 && p⟦2⟧ == 0, {{1, 0, 0}, {0, 1, 0}},
                  p⟦1⟧ == 0 && p⟦3⟧ == 0, {{1, 0, 0}, {0, 0, 1}},
                  p⟦2⟧ == 0 && p⟦3⟧ == 0, {{0, 1, 0}, {0, 0, 1}},
                  (* lattice point is on a coordinate plane *)
                  p⟦1⟧ != 0 && p⟦2⟧ != 0, {#, Cross[#, p]}&[{p⟦2⟧, -p⟦1⟧, 0}],
                  p⟦1⟧ != 0 && p⟦3⟧ != 0, {#, Cross[#, p]}&[{p⟦3⟧, 0, -p⟦1⟧}],
                  p⟦2⟧ != 0 && p⟦3⟧ != 0, {#, Cross[#, p]}&[{0, p⟦3⟧, -p⟦2⟧}],
                  (* lattice point is in generic position *)
                  True, {#, Cross[#, p]}&[{p⟦2⟧, -p⟦1⟧, 0}]]]
```

We are now adding some planes that guarantee that polygons are divided along the symmetry planes. A “symmetry unit” (for brevity, just called *unit* in the following) is given by the following domain $x \geq 0,\ y \geq 0,\ z \geq 0,\ z \geq x,\ x \geq y$. This is 1/48 of the whole space. The list `symmetrySlicingPlanes` contains the planes that bound the unit.

```
In[9]:= symmetrySlicingPlanes =
          Plane[{0, 0, 0}, #]& /@
              {{{1, 0, 0}, {0, 1, 0}}, {{1, 0, 0}, {0, 0, 1}},
               {{0, 1, 0}, {0, 0, 1}}, {{0, 0, 1}, {-1, 1, 0}},
               {{0, 0, 1}, {1, 1, 0}}, {{1, 0, 0}, {0, -1, 1}},
               {{1, 0, 0}, {0, 1, 1}}, {{0, 1, 0}, {-1, 0, 1}},
               {{0, 1, 0}, {1, 0, 1}}};
```

Here are the `symmetrySlicingPlanes`. The red, slightly sticking out polygons mark one symmetry unit. We will in the following concentrate on this unit and only later in the visualization part generate all other 47 units by reflection and rotation.

```
In[10]:= With[{ε = 0.05},
          Show[Graphics3D[{(* make polygons *)
              Polygon[{#⟦1⟧ - #⟦2, 1⟧ - #⟦2, 2⟧, #⟦1⟧ + #⟦2, 1⟧ - #⟦2, 2⟧,
                       #⟦1⟧ + #⟦2, 1⟧ + #⟦2, 2⟧, #⟦1⟧ - #⟦2, 1⟧ + #⟦2, 2⟧}]& /@
                                                          symmetrySlicingPlanes,
            {SurfaceColor[Hue[0]],(* lift a bit up *)
            Map[# + {ε, ε, 2ε}&, (* boundary of the unit cone *)
                {Polygon[{{0, 0, 0}, {0, 0, 1}, {1, 0, 1}}],
                 Polygon[{{0, 0, 0}, {0, 0, 1}, {1, 1, 1}}],
                 Polygon[{{0, 0, 0}, {1, 0, 1}, {1, 1, 1}}]},
                {-2}]}}], Boxed → False]];
```



`planes` is a list of all planes that contains the planes to the lattice points as well as the planes that bound the unit.

```
In[11]:= planes = Join[toPlane /@ latticePoints, symmetrySlicingPlanes];

In[12]:= Take[planes, 4]

Out[12]= {Plane[{-2, 0, 0}, {{0, 1, 0}, {0, 0, 1}}],
          Plane[{-3/2, -1, -1/2}, {{-2, 3, 0}, {-3, -2, 13}}],
          Plane[{-3/2, -1, 0}, {{-2, 3, 0}, {0, 0, 13}}],
          Plane[{-3/2, -1, 1/2}, {{-2, 3, 0}, {3, 2, 13}}]}
```

Next, we construct representations of the lines that are formed by the intersection of two planes. The presentation of the lines will be in the form `Line[`*onePointOfTheLine*`,` *lineDirection*`]`. We use here a two-argument version of `Line`, in distinction to the built-in `Line`, which takes one argument. The function `lineOnPlane[`*plane*$_1$`,` *plane*$_2$`]` calculates the intersection line which is located on *plane*$_1$, induced by its intersection with the plane *plane*$_2$.

```
In[13]:= (* three equations cannot be solved for four variables *)
         Off[Solve::"svars"];

         lineOnPlane[Plane[p_, {dir1_, dir2_}], Plane[q_, {d1_, d2_}]] :=
         Module[{eqs, sol, line, var, aux, P1, P2},
                If[(* are the planes parallel? *)
                   Length[DeleteCases[
                     RowReduce[{dir1, dir2, d1, d2}], {0, 0, 0}, {1}]] == 2, {},
                  (* calculate direction of the intersecting line *)
                   eqs = Thread[p + s dir1 + t dir2 == q + u d1 + v d2] ;
                   sol = Solve[eqs, {s, t, u, v}];
                   aux = p + s dir1 + t dir2 /. sol[[1]];
                   (* two points on the line *)
                   {P1, P2} = {aux /. {u -> 0, v -> 0}, aux /. {u -> 1, v -> 1}};
                   Line[P1, P2 - P1]]]
```

Here are two examples of the lines induced on the first plane.

```
In[16]:= lineOnPlane[planes[[1]], planes[[5]]]

Out[16]= Line[{-2, 1, -1}, {0, -20, 10}]

In[17]:= lineOnPlane[planes[[1]], planes[[11]]]

Out[17]= Line[{-2, 0, 1}, {0, -10, 0}]
```

`linesOnPlane1` is a list of all lines on the first plane that are induced by the other planes. We will take the first plane for exemplifying the further calculation steps.

```
In[18]:= (linesOnPlane1 = DeleteCases[
               Table[lineOnPlane[planes[[1]], planes[[i]]],
                     {i, 2, Length[planes]}], {}, {1}];) // Timing

Out[18]= {0.11 Second, Null}
```

We calculated 256 lines on the first plane.

```
In[19]:= Length[linesOnPlane1]

Out[19]= 256
```

```
In[20]:= Take[linesOnPlane1, 3]
```

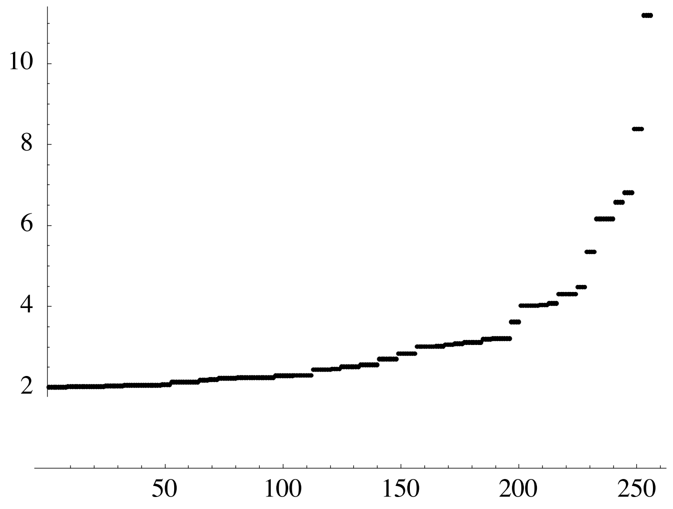
Out[20]= $\left\{\text{Line}\left[\left\{-2, -\frac{1}{4}, -\frac{1}{2}\right\}, \left\{0, -\frac{13}{2}, 13\right\}\right], \text{Line}\left[\left\{-2, -\frac{1}{4}, 0\right\}, \{0, 0, 13\}\right], \text{Line}\left[\left\{-2, -\frac{1}{4}, \frac{1}{2}\right\}, \left\{0, \frac{13}{2}, 13\right\}\right]\right\}$

To avoid superfluous computations, we will only keep the lines that are near the origin (some of the lines will have a minimum distance larger than `maxDist`). The function `lineDistanceSquare` calculates the minimum distance of a line to the origin.

```
In[21]:= lineDistanceSquare[Line[p_, dir_]] := #.#&[p - dir.p/dir.dir dir]
```

Here is the distribution of distances for all line segments from the first plane.

```
In[22]:= ListPlot[Sqrt[Sort[N[lineDistanceSquare /@ linesOnPlane1]]],
                  PlotRange -> All, AxesOrigin -> {0, 0}];
```



```
In[23]:= linesOnPlane1 = Select[linesOnPlane1, (* delete to far away lines *)
                                 lineDistanceSquare[#] < maxDist^2&];
```

This process reduced the number of lines by about 20%.

```
In[24]:= Length[linesOnPlane1]

Out[24]= 200
```

The next step is the calculation of the intersection of two lines. After checking if two lines intersect at all (by again using `RowReduce`), we will use `LinearSolve` to calculate the actual intersection point. Because `LinearSolve` does not have to deal with variables, it is slightly faster than `Solve`.

```
In[25]:= Off[LinearSolve::nosol];
         lineIntersection[Line[p1_, dir1_], Line[p2_, dir2_], δ_] :=
         Module[{ls},
             (* are the two lines the same? *)
                If[(Length[DeleteCases[RowReduce[{dir1, dir2}],
                                      {0, 0, 0}, {1}]] === 1) &&
                    (* check head *)
                    Head[LinearSolve[Transpose[{dir1}],
                              p2 - p1]] =!= LinearSolve,
                    Sequence @@ {},
              (* the actual intersection (if any) *)
                ls = LinearSolve[Transpose[{dir1, dir2}], p2 - p1];
                If[Head[ls] === LinearSolve, Sequence @@ {},
                   (* inside the sphere of interest? *)
                  If[#.# < δ^2, Point @ #, Sequence @@ {}]&[
                              p1 + ls[[1]] dir1]]]]
```

We collect all of the intersection points of all lines from the first plan in the list `intersectionsPlane1`. For efficiency, we again take into account only nearby points in maximum distance $2/4$ *maxDist*.

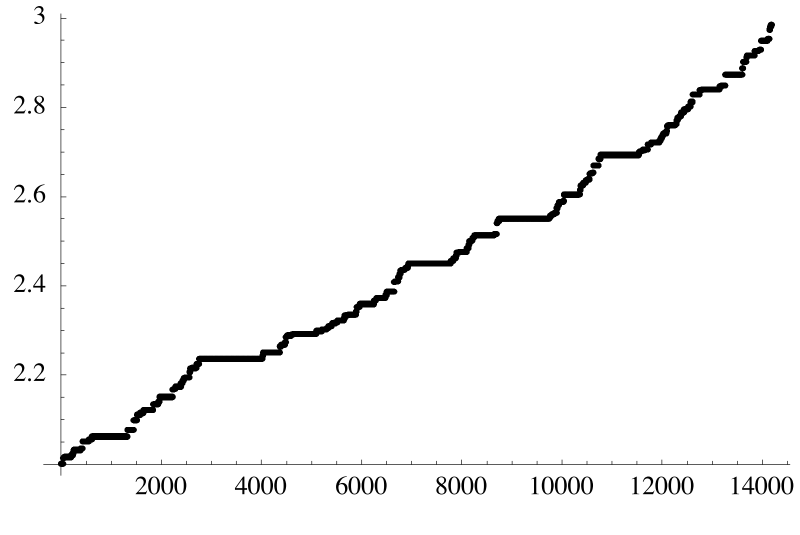
```
In[27]:= intersectionsPlane1 = DeleteCases[Table[If[i == j, Sequence @@ {},
             lineIntersection[linesOnPlane1[[j]], linesOnPlane1[[i]], 3/4 maxDist]],
                   {j, Length[linesOnPlane1]}, {i, Length[linesOnPlane1]}],
                                          {}, {1}]; // Timing

Out[27]= {8.28 Second, Null}
```

We found 14196 intersection points on the current plane.

```
In[28]:= Length[Flatten[intersectionsPlane1]]

Out[28]= 14196
```

Here is a distribution of the distance of the intersection points from the origin.

```
In[29]:= ListPlot[Sort[Sqrt[#[[1]].#[[1]]]& /@ N[Flatten[intersectionsPlane1]]],
                   PlotRange → All];
```



Given the points from a line, we form the line segments between the points. To do this, we fix one point and order the remaining points with respect to the distance from the chosen point.

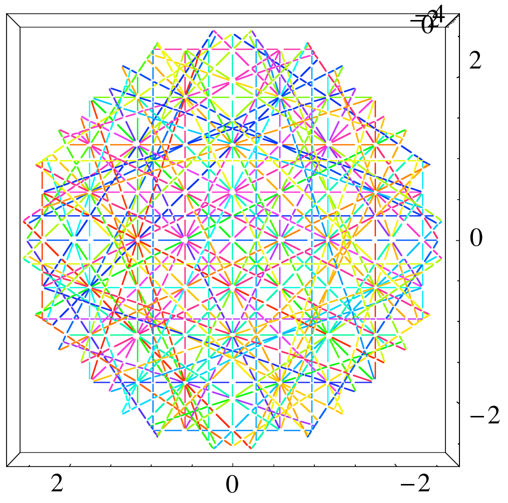
```
In[30]:= lineSegments[points_] :=
         Module[{points1 = Union[First /@ points],
                 distancesFromOrigin, startPoint},
                 (* distances from chosen point *)
                 distancesFromOrigin = #.#& /@ points1;
                 (* furthest point *)
                 startPoint = points1⟦Position[distancesFromOrigin,
                                    Max[distancesFromOrigin]]⟦1, 1⟧⟧;
                 (* sort points according to distance and form pairs *)
                 Line /@ Partition[Sort[points1,
                                   (#.#&[#1 - startPoint] <
                                    #.#&[#2 - startPoint])&], 2, 1]]
```

Here are all of the line segments of the first plane.

```
In[31]:= lineSegmentsPlane1 = lineSegments /@ intersectionsPlane1;

In[32]:= Length[Flatten[lineSegmentsPlane1]]

Out[32]= 4744
```

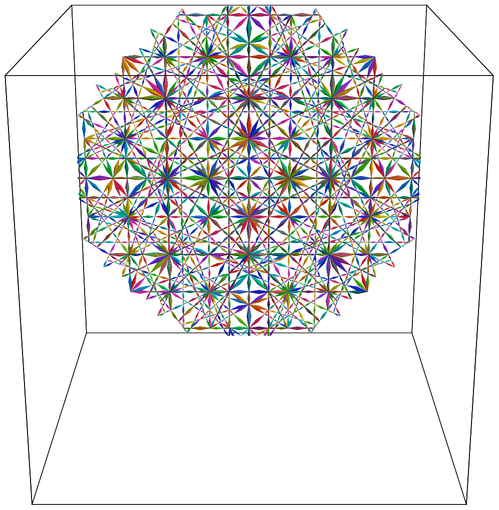
To show them, we contract them slightly.

```
In[33]:= contract[Line[{p1_, p2_}], f_] :=
         Module[{mp = (p1 + p2)}, Line[{mp + f(p1 - mp), mp + f(p2 - mp)}]]

In[34]:= Show[Graphics3D[{Hue[Random[]],
                           contract[#, 0.8]& /@ #}& /@ lineSegmentsPlane1],
              BoxRatios → {1, 1, 1}, Axes → True,
              (* view perpendicular *) ViewPoint → 4 latticePoints⟦1⟧];
```



We see the location of the line segments in 3D better when we use thin spindles instead of `Line` primitives.

```
In[35]:= makeSpindle[Line[{p1_, p2_}]] :=
         Module[{dir1, dir2, dir3, aux, mp = (p1 + p2)/2,
                  l = Sqrt[(p2 - p1). (p2 - p1)], n = 6},
                (* three orthogonal directions *)
                dir1 = n[p2 - p1];
                dir2 = n[Table[Random[], {3}]];
                dir2 = n[dir2 - dir1.dir2 dir1];
                dir3 = Cross[dir1, dir2];
                (* the spindle *)
                aux = Table[mp + l/12 (Cos[φ] dir2 + Sin[φ] dir3),
                             {φ, 0, 2. Pi, 2.Pi/n}];
                {Polygon[Append[#, p1]], Polygon[Append[#, p2]]}& /@
                              Partition[aux, 2, 1]]

In[36]:= Show[Graphics3D[{EdgeForm[], {SurfaceColor[Hue[Random[]]],
                          makeSpindle[#]}& /@ Flatten[lineSegmentsPlane1]}],
              BoxRatios → {1, 1, 1}, ViewPoint -> {3, 0, 1},
              PlotRange → 2.1{{-1, 1}, {-1, 1}, {-1, 1}}];
```



We see a high degeneracy in the sense that often more than two lines intersect at a given point. Here, it is calculated how often how many line segments meet at one point. (An odd number of lines meeting at a point is caused by points lying on the boundary.)

```
In[37]:= {#[[1]], Length[#]}& /@ Split[Sort[Length /@ Split[Sort[
            Join[#, Reverse /@ #]&[First /@ Flatten[lineSegmentsPlane1]]],
                                         #1⟦1⟧ === #2⟦1⟧&]]]

Out[37]= {{2, 24}, {3, 40}, {4, 636}, {5, 20}, {6, 364},
          {7, 12}, {8, 188}, {9, 8}, {10, 72}, {12, 40}, {14, 28},
          {16, 21}, {18, 8}, {22, 8}, {24, 8}, {28, 4}, {32, 4}, {38, 4}}
```

We keep exactly one (directed) copy of each line segment to form polygons from the line segments.

```
In[38]:= allLineSegmentsPlane1 =
          Join[#, Map[Reverse, #, {2}]]&[
          Flatten[(* give name *) directedLineSegmentsPlane1 =
            Union[Map[Sort, Flatten[lineSegmentsPlane1], {2}]]]];
```

A fast method for the determination of point-line connection is implemented in the following input. It uses hashing techniques of the built-in function `Set`, instead of repeated calls to `Select` for every point and line segment. Here, we introduce the globally visible variable `linesFromPoint`.

```
In[39]:= makePointsFromLines[lineSegments_] :=
         Module[{allLineSegments, allCrossingPoints},
                (* each line segment in each direction must appear once *)
                 allLineSegments = Join[#,
                      Map[Reverse, #, {2}]]&[Flatten[lineSegments]];
                 (* all crossing points *)
                 allCrossingPoints = Point /@ Union[
                           Flatten[First /@ allLineSegments, 1]];
                (* which line segments start from which point *)
               Clear[linesFromPoint];
               (* create definitions for linesFromPoint *)
               (linesFromPoint[#] = {})& /@ allCrossingPoints;
               (linesFromPoint[Point[#[[1, 1]]]] =
                     {linesFromPoint[Point[#[[1, 1]]]], #})& /@
                                                 allLineSegments;
               (linesFromPoint[#] = Flatten[linesFromPoint[#]])& /@
                                            allCrossingPoints;]

In[40]:= makePointsFromLines[directedLineSegmentsPlane1] // Timing

Out[40]= {0.38 Second, Null}
```

Here are the current definitions for the global variable `linesFromPoint`.

```
In[41]:= Take[DownValues[linesFromPoint], 2]
```

Out[41]= $\{\text{HoldPattern}[\text{linesFromPoint}[\text{Point}[\{-2, -\frac{11}{5}, -\frac{1}{5}\}]]] \mapsto$
$\{\text{Line}[\{\{-2, -\frac{11}{5}, -\frac{1}{5}\}, \{-2, -\frac{13}{6}, -\frac{1}{6}\}\}],$
$\text{Line}[\{\{-2, -\frac{11}{5}, -\frac{1}{5}\}, \{-2, -\frac{17}{8}, -\frac{1}{4}\}\}]\},$
$\text{HoldPattern}[\text{linesFromPoint}[\text{Point}[\{-2, -\frac{11}{5}, \frac{1}{5}\}]]] \mapsto$
$\{\text{Line}[\{\{-2, -\frac{11}{5}, \frac{1}{5}\}, \{-2, -\frac{13}{6}, \frac{1}{6}\}\}],$
$\text{Line}[\{\{-2, -\frac{11}{5}, \frac{1}{5}\}, \{-2, -\frac{17}{8}, \frac{1}{4}\}\}]\}\}$

The function `clockwiseSort` now sorts all line segments originating from a given point in clockwise order. To avoid recalculations we dynamically create definitions with `mostLeftSegment`. `mostLeftSegment[Line[`*point*$_1$, *intersectionPoint*`], Line[`*intersectionPoint*, *point*$_2$`]]` gives the left-most line segments with respect to the line `Line[`*point*$_1$, *intersectionPoint*`]` at the point *intersectionPoint*.

```
In[42]:= clockwiseSort[Point[p_], lines_, plane_] :=
         Module[{lines1, sorted},
                (* two orthogonal directions *)
                {dir1, dir2} = N[#/Sqrt[#.#]& /@ plane[[2]]];
                (* angles of all line segments *)
                lines1 = {#, ArcTan[(#[[1, 2]] - #[[1, 1]]).dir1,
                                    (#[[1, 2]] - #[[1, 1]]).dir2]}& /@ lines;
               (* sort angle list *)
                sorted = First /@ Sort[lines1, #1[[2]] < #2[[2]]&];
               (* create definitions for mostLeftSegment *)
         Apply[(mostLeftSegment[Map[Reverse, #1, {1}]] = #2)&,
                     Partition[Append[sorted, First[sorted]], 2, 1], {1}]]
```

```
In[43]:= Timing[clockwiseSort[#[[1, 1, 1]], #[[2]], planes[[1]]]& /@
                                         DownValues[linesFromPoint];]

Out[43]= {1.17 Second, Null}
```

Here are the current definitions for the global variable `mostLeftSegment`.

```
In[44]:= Take[DownValues[mostLeftSegment], 2]
```

Out[44]= $\{\text{HoldPattern}[\text{mostLeftSegment}[\text{Line}[\{\{-2, -\frac{11}{5}, -\frac{1}{5}\}, \{-2, -\frac{13}{6}, -\frac{1}{6}\}\}]]] :\!\to \text{Line}[\{\{-2, -\frac{13}{6}, -\frac{1}{6}\}, \{-2, -2, -\frac{1}{4}\}\}],$
$\text{HoldPattern}[\text{mostLeftSegment}[\text{Line}[\{\{-2, -\frac{11}{5}, -\frac{1}{5}\}, \{-2, -\frac{17}{8}, -\frac{1}{4}\}\}]]] :\!\to \text{Line}[\{\{-2, -\frac{17}{8}, -\frac{1}{4}\}, \{-2, -2, -\frac{1}{3}\}\}]\}$

Now, we can implement a function `makePolygon` that forms recursively polygons from the line segments in a plane. To do this, we start at an intersection of two lines, move along the line until the next intersection, there we turn into the left-most next line until the next intersection comes, there we turn into the left-most, and so on until we again arrive at the starting point.

```
In[45]:= (* add the next left-most segments *)
         makePolygon[Polygon[s___, lastSegment_]] :=
             makePolygon[Polygon[s, lastSegment, mostLeftSegment[lastSegment]]]

         (* in case the last point is equal to the first one -- we are done *)
         makePolygon[Polygon[firstSegment_, s___, firstSegment_]] :=
          ((lineSegmentAlreadyUsedQ[#] = True)& /@ {firstSegment, s};
                                  Polygon[#[[1, 1]]& /@ {firstSegment, s}])
```

The calculation of the polygons can now be done quickly.

```
In[50]:= (allPolysPlane1 = Flatten[
             If[TrueQ[lineSegmentAlreadyUsedQ[#]], {},
                makePolygon[Polygon[#]]]& /@ allLineSegmentsPlane1];) // Timing

Out[50]= {0.52 Second, Null}
```

The first plane contains 2033 polygons at this point.

```
In[51]:= allPolysPlane1 // Length

Out[51]= 2033
```

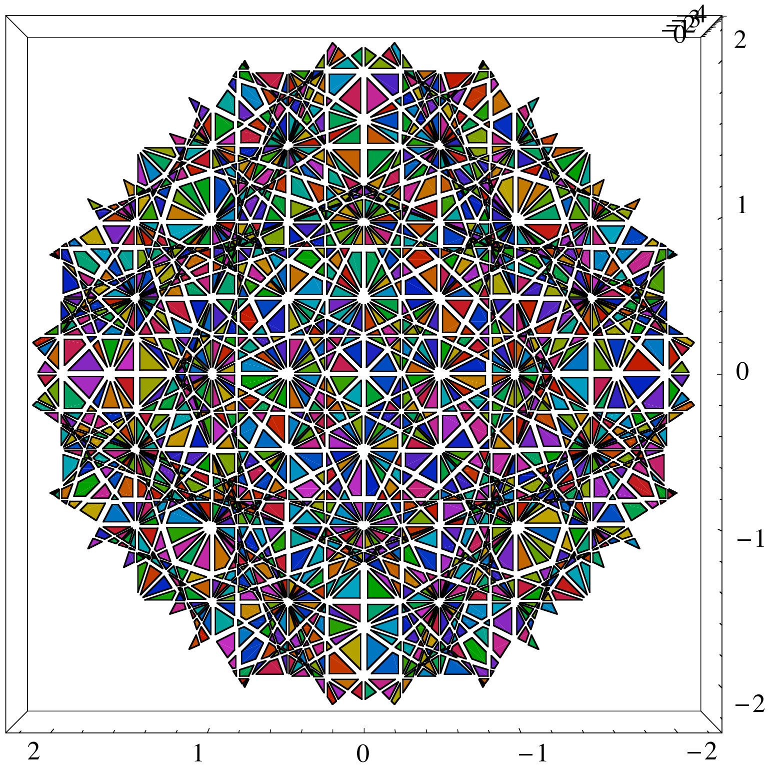
We also have the outer polygon in our current list. It can be easily found because it has the most vertices.

```
In[52]:= deleteOuterPolygon[polys_] :=
          Delete[polys, Position[#, Max[#]][[1, 1]]&[Length[#[[1]]]& /@ polys]

         deleteOuterPolygon[{}] := {}

In[55]:= allPolysPlane1 = deleteOuterPolygon[allPolysPlane1];
```

Here are the polygons from our working plane.

```
In[56]:= contract[Polygon[l_], f_] :=
          With[{mp = Plus @@ l/Length[l]}, Polygon[mp + f(# - mp)& /@ l]]
```

```
In[57]:= Show[Graphics3D[{SurfaceColor[Hue[Random[]]],
                          contract[#, 0.7]}& /@ allPolysPlane1],
              BoxRatios → {1, 1, 1}, Axes → True,
               ViewPoint → 4 latticePoints[[1]]];
```

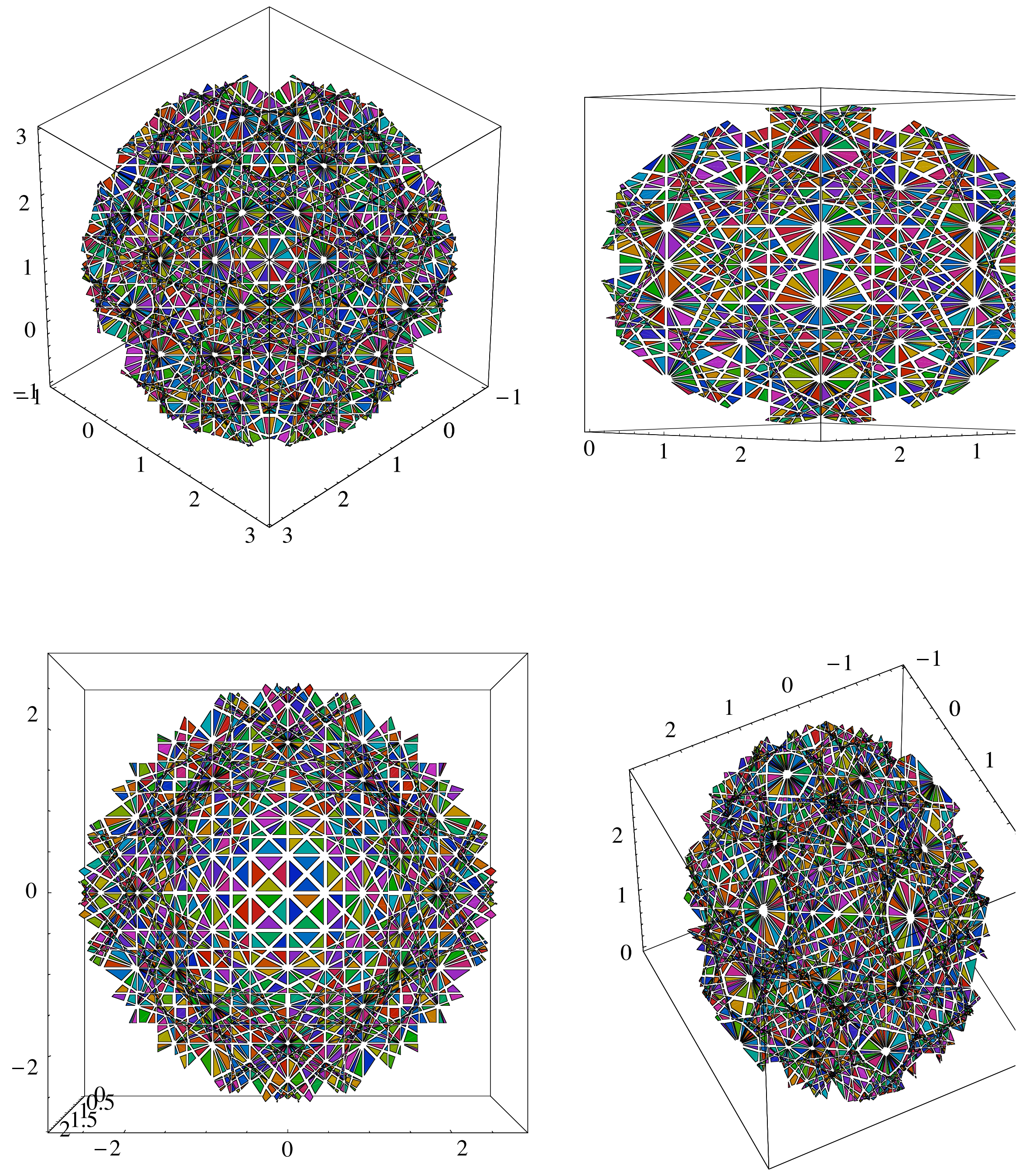


We now put all of the functions above together into a function `tessellatePlane`. `tessellatePlane[`*plane*`,` *planes*`,` δ`]` will give a list of all of the polygons in plane *plane* which are induced by intersections with the planes *planes* within a maximal distance δ.

```
In[58]:= tessellatePlane[plane_, planes_, δ_] :=
                   tessellatePlane[plane, planes, δ] =
         Module[{linesOnThePlane, planeIntersections,
         directedPlaneLineSegments, allPlaneLineSegments},
         (* the lines formed by the intersection with the other planes *)
         linesOnThePlane = Select[DeleteCases[
                  Table[lineOnPlane[plane, planes[[i]]],
                       {i, Length[planes]}], {}, {1}],
                                lineDistanceSquare[#] <= δ^2&];
         (* the intersection points inside the planes *)
          planeIntersections =
           DeleteCases[Table[If[i === j, Sequence @@ {},
                 lineIntersection[linesOnThePlane[[j]], linesOnThePlane[[i]], δ]],
                           {j, Length[linesOnThePlane]},
                           {i, Length[linesOnThePlane]}], {}, {1}];
         (* the line segments inside the planes *)
         directedPlaneLineSegments = Union[Map[Sort,
             Flatten[lineSegments /@ planeIntersections], {2}]];
         (* line segments in both directions *)
         allPlaneLineSegments = Join[#, Map[Reverse, #, {2}]]&[
                                 Flatten[directedPlaneLineSegments]];
         (* the polygons inside the planes *)
         Clear[mostLeftSegment, lineSegmentAlreadyUsedQ];
         makePointsFromLines[directedPlaneLineSegments];
         clockwiseSort[#[[1, 1, 1]], #[[2]], plane]& /@
                                  DownValues[linesFromPoint];
         allPolys = Flatten[If[TrueQ[lineSegmentAlreadyUsedQ[#]], {},
                             makePolygon[Polygon[#]]]& /@
                                             allPlaneLineSegments];
         (* delete the outermost polygon *)
         deleteOuterPolygon[allPolys]]
```

Here are four more examples of tessellated planes. Depending on the direction of the lattice point, the resulting tessellation of the plane has different symmetries. We also count the number of polygons inside one plane to get an idea of the total number of polygons to deal with in the next step.

```
In[59]:= (* auxiliary function for generating a picture *)
         tessellatedPlanePicture[latticePoint_, δ_, opts___] :=
         Module[{plane = toPlane[latticePoint], polys},
                (* the polygons to display *)
                polys = tessellatePlane[plane, DeleteCases[planes, plane], δ];
                 (* the picture *)
                 Show[Graphics3D[{SurfaceColor[Hue[Random[]]],
                                  EdgeForm[Thickness[0.0001]],
                                  contract[#, 0.7]}& /@ polys],
                      opts, BoxRatios → {1, 1, 1}, Axes → True,
                      (* view from perpendicular direction *)
                      ViewPoint → 3 latticePoint]]

In[61]:= Function[points, Show[GraphicsArray[
         Block[{$DisplayFunction = Identity},
               tessellatedPlanePicture[#, 3]& /@ points]]]] /@
               {{{2, 2, 2}, {3, 3, 0}}, {{2, 0, 0}, {1, 2, 3}}};
```



Here are the number of polygons that appear in the last graphics.

```
In[62]:= % /. {gr3d_Graphics3D :> Count[gr3d, _Polygon, Infinity],
               GraphicsArray -> List}

Out[62]= %61
```

2. Construction of the Polygons in the Symmetry Unit

We saw that we get a couple of thousand polygons per plane. Taking into account that we have to deal with a few hundred planes, this would give a very large number of polygons. It is now time to take the symmetry of the lattice into account. We only calculate all polygons inside the unit described above (and later on, we rotate and mirror the polygons that form the Brillouin zones into the other 47 units). Inside the symmetry unit, we have just 15 different points to consider.

```
In[63]:= selectedLatticePoints =
         Select[latticePoints, (#⟦1⟧ >= 0 && #⟦2⟧ >= 0 &&
                                #⟦3⟧ >= 0 && #⟦1⟧ >= #⟦2⟧ &&
                                #⟦3⟧ >= #⟦1⟧)&]

Out[63]= {{0, 0, 1}, {0, 0, 2}, {0, 0, 3}, {0, 0, 4}, {1, 0, 1},
          {1, 0, 2}, {1, 0, 3}, {1, 1, 1}, {1, 1, 2}, {1, 1, 3},
          {2, 0, 2}, {2, 0, 3}, {2, 1, 2}, {2, 1, 3}, {2, 2, 2}}

In[64]:= Length[selectedLatticePoints]

Out[64]= 15
```

This does, of course, not mean that we only have to calculate the polygons in the planes formed by lattice points inside the unit. Other planes might also contribute polygons into the unit (they intersect with the planes associated with the points from the unit). But the plane tessellations of lattice points outside the unit are the same as for the equivalent point inside the unit. The function `transformIntoUnit` gives the equivalent point in the unit as well as the matrix, which rotates the point to the equivalent point. Later, we will use the inverse of this matrix to rotate the polygons back into the other units. `transform·IntoUnit` works recursively until the point is within the first unit.

```
In[65]:= transformIntoUnit[{x_, y_, z_}, mat_] :=
             transformIntoUnit[{-x, y, z},
                         {{-1, 0, 0}, {0, 1, 0}, {0, 0, 1}}.mat] /; x < 0

         transformIntoUnit[{x_, y_, z_}, mat_] :=
             transformIntoUnit[{x, -y, z},
                         {{1, 0, 0}, {0, -1, 0}, {0, 0, 1}}.mat] /; y < 0

         transformIntoUnit[{x_, y_, z_}, mat_] :=
             transformIntoUnit[{x, y, -z},
                         {{1, 0, 0}, {0, 1, 0}, {0, 0, -1}}.mat] /; z < 0

         transformIntoUnit[{x_, y_, z_}, mat_] :=
             transformIntoUnit[{y, x, z},
                         {{0, 1, 0}, {1, 0, 0}, {0, 0, 1}}.mat ] /; y > x

         transformIntoUnit[{x_, y_, z_}, mat_] :=
             transformIntoUnit[{z, y, x},
                         {{0, 0, 1}, {0, 1, 0}, {1, 0, 0}}.mat ] /; x > z
```

The function `tessellatePlane1` is built on top of the function `tessellatePlane` and uses the memorized tessellations of `tessellatePlane`. It takes a tessellated plane from inside the unit and rotates it into the position of the lattice point under consideration.

```
In[70]:= tessellatePlane1[latticePoint_, δ_] :=
    Module[{latticePointInsideUnit, mat, rotationMatrix,
            currentPlane, polys},
            (* the equivalent point inside the unit and
               the corresponding rotation matrix *)
           {latticePointInsideUnit, mat} =
            List @@ transformIntoUnit[latticePoint, IdentityMatrix[3]];
           (* the rotation matrix from the point inside the
              unit to the original point *)
          rotationMatrix = Inverse[mat];
           (* the tessellation of the point inside the unit *)
          currentPlane = toPlane[latticePointInsideUnit];
          polys = tessellatePlane[toPlane[latticePointInsideUnit],
                 DeleteCases[planes, currentPlane], δ];
           (* rotate all polygons inside the original position *)
            If[polys =!= {}, Map[rotationMatrix.#&, polys, {-2}], {}]]
```

The function `inUnitQ` determines if a given polygon is inside the first symmetry unit.

```
In[71]:= (* for a single point *)
    inUnitQ[{x_, y_, z_}] := x >= 0 && y >= 0 && z >= 0 &&
                             x >= y && z >= x
    (* for a whole polygon *)
    inUnitQ[Polygon[l_]] := And @@ (inUnitQ /@ l)
```

We now carry out the tessellation of all planes and keep only the polygons inside the unit and inside a distance `maxDist/2`. This is the most time-consuming operation of the whole construction. With the currently used parameters, this will take less than $2\frac{1}{2}$ minutes on a 2 GHz computer.

```
In[75]:= (polygonsInUnit = Select[tessellatePlane1[#, maxDist/2],
                             inUnitQ]& /@ latticePoints); // Timing

Out[75]= {67.55 Second, Null}
```

After dealing with all 256 lattice points, we had to do the hard work—the tessellation of a plane—only for a small fraction of them.

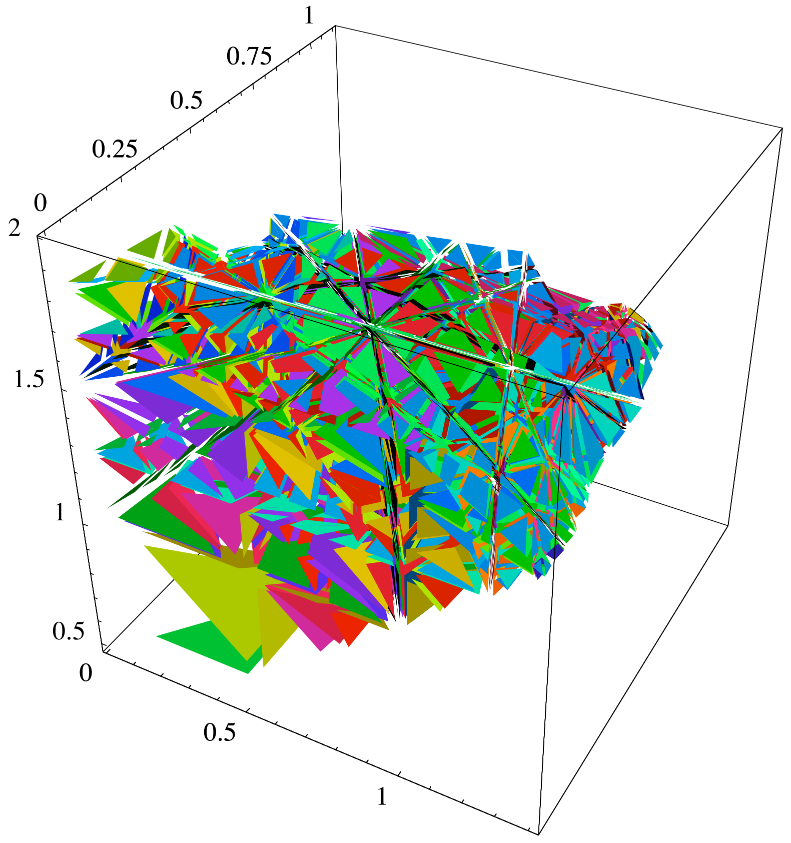
```
In[76]:= Length[DownValues[tessellatePlane]]

Out[76]= 20
```

About 20000 polygons were calculated.

```
In[77]:= Count[DownValues[tessellatePlane], _Polygon, Infinity]

Out[77]= 19540
```

Here a picture of the shrunken polygons inside the unit is shown; all polygons originating from one plane are shown in the same color.

```
In[78]:= Show[Graphics3D[{EdgeForm[], SurfaceColor[Hue[Random[]]],
                        (contract[#, 0.66]&) /@ #}& /@ polygonsInUnit],
              BoxRatios -> {1, 1, 1}, Axes -> True, PlotRange -> All];
```



Inside the unit, we now have 2650 polygons. If we did not take symmetry into account, we would have to deal with nearly 130000 polygons!

```
In[79]:= Length[polygonsInUnit = Flatten[polygonsInUnit]]

Out[79]= 2650
```

3. Build the Brillouin Zones from the Polygons

To build the Brillouin zones, we need a polygon to start with. We start with one that touches the *z*-axis and is currently the "innermost" one. The function `selectPolygonsOnZAxis` selects all polygons that have a point at the *z*-axis.

```
In[80]:= selectPolygonsOnZAxis[polygonsInUnit_] :=
         Select[Select[polygonsInUnit, MemberQ[#⟦1⟧,  {0, 0, _}]&],
                MemberQ[DeleteCases[#⟦1⟧, {0, 0, _}], {_, 0, _}]&];
```

We now have 27 such polygons.

```
In[81]:= (polygonsOnZAxis =
              selectPolygonsOnZAxis[polygonsInUnit]) // Length

Out[81]= 27
```

It can happen that some polygons share a point at the *z*-axis. In this case, the currently "innermost" polygon is the one that points the most downward in the *x*,*z*-plane or, if some polygons point in the same direction, the direction in the *y*-direction. The function `zValueAndAngle` determines the point at the *z*-axis and the two mentioned angles.

```
In[82]:= zValueAndAngle[Polygon[l_]] :=
             Module[{zAxisPoint, zValue, xzPlanePoint,
                     α, mp, dir1, dir2, dir3, β},
                    (* first look at the z value of the point on the z-axis *)
                    zAxisPoint = Cases[l, {0, 0, _}]⟦1⟧;
                    zValue = zAxisPoint⟦3⟧;
                    (* second look at the height of the
                       point on the x-z plane *)
                    xzPlanePoint = Cases[l, {_?(# =!= 0&), 0, _}]⟦1⟧;
                    α = ArcTan[xzPlanePoint⟦1⟧, xzPlanePoint⟦3⟧ - zValue];
                    (* in case polygons have the same edge in the x,z-plane,
                       compare the angle of the midpoint with respect to the edge *)
                    mp = Plus @@ l/Length[l];
                    dir1 = n[zAxisPoint - xzPlanePoint];
                    dir2 = {0, 1, 0};
                    dir3 = Cross[dir2, dir1];
                    β = ArcTan[Expand[(mp - xzPlanePoint).dir2],
                               Expand[(mp - xzPlanePoint).dir3]];
                    (* return data *)
                    {zValue, α, β}]
```

Now, we sort the polygons `polygonsOnZAxis` with respect to the sorting function `polygonsOnZAxisSortedQ`.

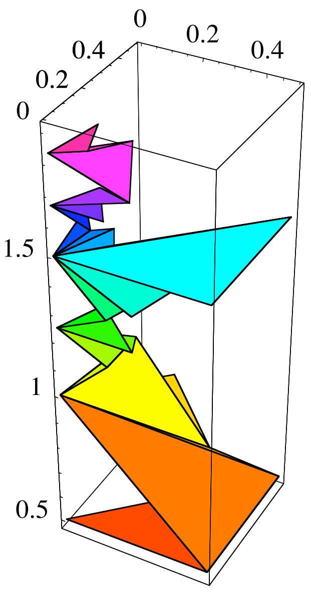
```
In[83]:= polygonsOnZAxisSortedQ[{_Polygon, {z1_, α1_, β1_}},
                               {_Polygon, {z2_, α2_, β2_}}] :=
             Which[(* compare z values *) z1 < z2, True, z1 > z2, False,
                   (* compare α angles *) α1 < α2, True, α1 > α2, False,
                   (* compare β angles *) β1 < β2, True, β1 > β2, False]

In[84]:= sortPolygonsOnZAxis[polygonsOnZAxis_] :=
             First /@ Sort[{#, zValueAndAngle[#]}& /@ polygonsOnZAxis,
                                                  polygonsOnZAxisSortedQ]
```

The list `sortedStartingPolygons` contains the 27 polygons to start with in the recursive building process of Brillouin zones.

```
In[85]:= sortedStartingPolygons = sortPolygonsOnZAxis[polygonsOnZAxis];
```

Here, we color the sorted polygons on the *z*-axis from red to purple, with the red polygons being the ones nearest to the origin.

```
In[86]:= Show[Graphics3D[MapIndexed[{Hue[#2[[1]]/30], #1}&,
                         sortedStartingPolygons]],
              Lighting → False, Axes → True, PlotRange → All];
```



For building the Brillouin zones, we start with a polygon from the list `sortedStartingPolygons` and add at all edges that are not on the boundary of the symmetry unit, the next innermost one. To have fast access to the polygons that meet at a given edge, we use the `Set`-based hashing technique from above and build a function `polygonsFromEdge`.

```
In[87]:= (* subsidiary definition of a function that returns
            a list of directed edges of a given polygon *)
         edges[Polygon[l_]] := Edge /@ Sort /@
                         Partition[Append[l, First[l]], 2, 1]

In[89]:= edgesAndPolygons[polys_] :=
         Module[{allEdges},
         (* (all oriented) edges *)
         allEdges = Union[Flatten[edges /@ polys]] ;
         (* which polygon is connected to which edge *)
         Clear[polygonsFromEdge];
         (polygonsFromEdge[#] = {})& /@ allEdges;
         Function[p, Map[(polygonsFromEdge[#] = {polygonsFromEdge[#], p})&,
                          edges[p], {1}]] /@ polys;
         (polygonsFromEdge[#] = Flatten[polygonsFromEdge[#]])& /@ allEdges;]

In[90]:= edgesAndPolygons[polygonsInUnit]; // Timing

Out[90]= {1.04 Second, Null}
```

As a result, the function `polygonsFromEdge` now has 2324 definitions.

```
In[91]:= DownValues[polygonsFromEdge] // Length

Out[91]= 2324
```

Here are some of these definitions.

In[92]:=
```
Take[DownValues[polygonsFromEdge], 3]
```

Out[92]=

$$\{\text{HoldPattern}[\text{polygonsFromEdge}[\text{Edge}[\{\{0, 0, \tfrac{1}{2}\}, \{\tfrac{1}{2}, 0, \tfrac{1}{2}\}\}]]] :\to \{\text{Polygon}[\{\{0, 0, \tfrac{1}{2}\}, \{\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}\}, \{\tfrac{1}{2}, 0, \tfrac{1}{2}\}\}]\},$$
$$\text{HoldPattern}[\text{polygonsFromEdge}[\text{Edge}[\{\{0, 0, \tfrac{1}{2}\}, \{\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}\}\}]]] :\to \{\text{Polygon}[\{\{0, 0, \tfrac{1}{2}\}, \{\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}\}, \{\tfrac{1}{2}, 0, \tfrac{1}{2}\}\}]\},$$
$$\text{HoldPattern}[\text{polygonsFromEdge}[\text{Edge}[\{\{0, 0, 1\}, \{\tfrac{1}{6}, 0, \tfrac{7}{6}\}\}]]] :\to \{\text{Polygon}[\{\{\tfrac{1}{6}, 0, \tfrac{7}{6}\}, \{0, 0, 1\}, \{\tfrac{1}{6}, \tfrac{1}{6}, \tfrac{7}{6}\}\}]\}\}$$

Here is a list of the number of polygons that have one edge in common.

In[93]:=
```
{#[[1]], Length[#]}& /@ Split[Sort[Length[Last[#]]& /@
                                        DownValues[polygonsFromEdge]]]
```

Out[93]=
```
{{1, 180}, {2, 362}, {3, 189}, {4, 1274},
  {5, 37}, {6, 272}, {10, 7}, {12, 1}, {14, 2}}
```

Now, we must implement a function `nextPolygon` that, given a polygon from a Brillouin zone and one of its free edges, selects still available polygons from among all of the “innermost” ones. The function `outsideNormal` is an auxiliary function determining the outside normal of a given polygon (outside meaning pointing away from the origin). We could have saved this information from the beginning, but its recalculation is fast and simple, so we prefer to recalculate it instead dealing with more complicated and more memory-consuming data structures.

In[94]:=
```
outsideNormal[Polygon[l_]] :=
     Module[{mp = Plus @@ l/Length[l], dir},
            dir = Cross[l⟦1⟧ - mp, l⟦2⟧ - mp];
            If[dir.mp < 0, dir = -dir];
            (* rewrite in nice form *)
            #/GCD @@ Abs[#]&[(LCM @@ Denominator[dir]) dir]]
```

In[95]:=
```
nextPolygon::noPolygonsLeft = "Out of polygons at edge `1`.";
```

```
In[96]:= (* in case not enough polygons were calculated *)
         nextPolygon::noPolygonsLeft = "Out of polygons at edge `1`.";

         nextPolygon[poly:Polygon[l_], edge:Edge[{p1_, p2_}]] :=
         Module[{possiblePolys, dir1, dir2, dir3,
                  polyMps, polygonsAndAngles},
                 (* all still not used polygons at the current edge *)
                 possiblePolys = polygonsFromEdge[edge];
                 (* a crude safety hack for too high orders *)
                 If[possiblePolys === {},
                    Message[nextPolygon::noPolygonsLeft, edge]; Abort[]];
                 (* make three orthogonal directions *)
                 dir1 = n[p2 - p1];
                 mp = Plus @@ l/Length[l];
                 dir2 = mp - p1;
                 dir2 = -Together[n[dir2 - dir1.dir2 dir1]];
                 dir3 =  Together[n[outsideNormal[poly]]];
                 (* the relative orientation of the polygons *)
                 polyMps = {#, Plus @@ #⟦1⟧/Length[#⟦1⟧]}& /@ possiblePolys;
                 polygonsAndAngles = {#⟦1⟧,
                           ArcTan[Together[(#⟦2⟧ - p1).dir2],
                                  Together[(#⟦2⟧ - p1).dir3]]}& /@ polyMps;
                 (* the “next” polygon *)
                 Sort[polygonsAndAngles, #1⟦2⟧ < #2⟦2⟧&]⟦1, 1⟧]
```

Now, we can implement the last step: the actual recursive building process for the Brillouin zones. The function `buildBrillouinZone` contains a list of lists. Each sublist is of the form {*polygon*, *freeEdgesOfThisPolygon*}. As long as free edges exist, we add them until the polygons fill the solid angle of the symmetry unit.

```
In[99]:= (* remove double copies of free edges *)
         buildBrillouinZone[l_] :=
         Module[{cond = First /@ Cases[Function[p,
             {#, Count[p, #]}& /@ Union[p]][Flatten[Last /@ l]], {_, 2}]},
              buildBrillouinZone[DeleteCases[l,
                                 Alternatives @@ cond, Infinity]] /;
                                                       cond =!= {}]

         (* add polygons to the free edges of a particular polygon *)
         buildBrillouinZone[{a___, b:{poly_, freeEdges_?(# =!= {}&)}, c___}] :=
             buildBrillouinZone[{a, {First[b], {}}, addPolys[b], c}]

         (* no free edges present anymore; take out the polygons *)
         buildBrillouinZone[l:{{_, {}}..}] := First /@ l
```

We do not have to add polygons to edges that lie on the boundary of the unit. The function `outerEdgeQ` tests if a given edge is on a boundary.

```
In[105]:= outerEdgeQ[Edge[{{x1_, y1_, z1_}, {x2_, y2_, z2_}}]] :=
            (y1 == y2 == 0) || ((x1 == y1) && (x2 == y2)) ||
                               ((x1 == z1) && (x2 == z2))
```

After a polygon has been added to the current Brillouin zone, it must be taken out from the polygons memorized in `polygonsFromEdge`. The function `upDatePolygonsFromEdgeDefinitions` is carrying out this updating process.

```
In[106]:= upDatePolygonsFromEdgeDefinitions[poly_] :=
           (polygonsFromEdge[#] =
              DeleteCases[polygonsFromEdge[#], poly])& /@ edges[poly]
```

The main work in `buildBrillouinZone` is done by the function `addPolys`. For a given polygon and its free edges, it determines which polygons are the next innermost neighboring polygons and properly updates all lists of free edges, polygons, and `polygonsFromEdge`.

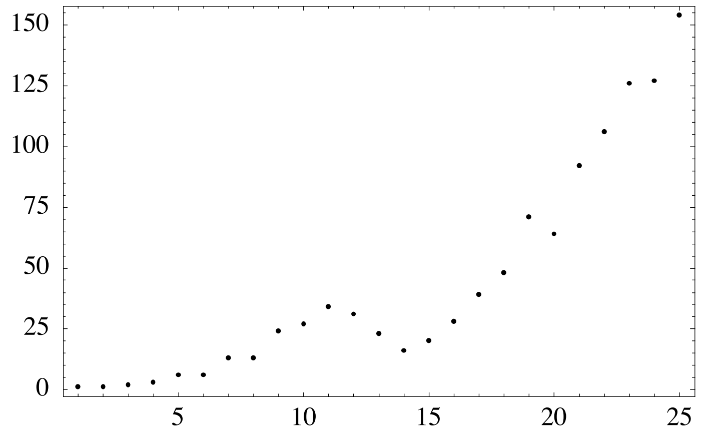
```
In[107]:= addPolys[{poly_, freeEdges_}] :=
          Module[{newPolygons, newEdges, doubleEdges, newFreeEdges, outerEdges},
          (* the polygons innermost and having edge in common *)
          newPolygons = Union[nextPolygon[poly, #]& /@ freeEdges];
          (* the edges of these new polygons *)
          newEdges = edges /@ newPolygons;
          (* some of the new polygons might share a new edge *)
          doubleEdges = First /@ Cases[
              Function[l, {#, Count[l, #]}& /@ Union[l]][
                                         Flatten[newEdges, 1]], {_, 2}];
          (* the old free edge is no longer a free edge *)
          newFreeEdges = DeleteCases[newEdges,
                                         Alternatives @@ freeEdges, {2}];
          (* two of the new polygons might have an edge in common *)
          newFreeEdges = DeleteCases[newFreeEdges,
                                         Alternatives @@ doubleEdges, {2}];
          (* some of the new edges might be outer edges *)
          outerEdges = Union[Cases[newFreeEdges, _?outerEdgeQ, {2}]];
          newFreeEdges = DeleteCases[newFreeEdges,
                                        Alternatives @@ outerEdges, {2}];
          (* now update the edgePolygon definitions *)
          upDatePolygonsFromEdgeDefinitions /@ newPolygons;
          (* return new borderline *)
          Sequence @@ Transpose[{newPolygons, newFreeEdges}]]
```

Now, we can finally calculate the polygons forming the Brillouin zones. The polygons we have stored in `polygonsInUnit` allow us to calculate the first 25 Brillouin zones (its polygons inside the unit, respectively). `brillouinZonePolygons`⋱ `SC[`*i*`]` is a list of the polygons of the *i*th Brillouin zone of the simple cubic lattice.

```
In[108]:= Off[$MaxExtraPrecision::meprec];
          Do[(* nearest polygon *)
             startPoly = sortedStartingPolygons[[i]];
             (* free edges of the nearest polygon *)
             freeStartEdges = Complement[#,
                                Select[#, outerEdgeQ]]&[edges[startPoly]];
             upDatePolygonsFromEdgeDefinitions[startPoly];
             brillouinZonePolygonsSC[i] =
                    buildBrillouinZone[{{startPoly, freeStartEdges}}],
             {i, 25}] // Timing

Out[109]= {7.63 Second, Null}
```

Here are the number of polygons of the *i*th Brillouin zone in the unit shown.

```
In[110]:= ListPlot[Table[{k, Length[brillouinZonePolygonsSC[k]]}, {k, 25}],
                   PlotRange -> All, Frame -> True, Axes -> False];
```



4. Visualize the Brillouin Zones

Before making some pictures of Brillouin zones, let us check if the Brillouin zones calculated at the end of the last subsection really have the expected volume. The function `volume` calculates the volume of the polyhedron bounded outside by the polygons of the Brillouin zones [9✱].

```
In[111]:= (* all together there are 48 units *)
          volume[polys_List] := 48 Plus @@ (volume /@ polys);

          (* the basic case *)
          volume[Polygon[{p1_, p2_, p3_}]] :=
                         volume[TrianglePyramid[{p1, p2, p3}]];

          (* split polygons with more than three vertices into triangles *)
          volume[Polygon[l_]] :=
              Module[{mp = Plus @@ l/Length[l]},
                  Plus @@ (volume[TrianglePyramid[Append[#, mp]]]& /@
                             Partition[Append[l, First[l]], 2, 1])]

          (* the volume of a pyramid with triangular base *)
          volume[TrianglePyramid[{p1_, p2_, p3_}]] :=
                  Abs[Cross[p1 - p2, p1 - p3].(p1 + p2 + p3)]/18
```

Here, the volumes are calculated. We see that all Brillouin (in the sense of the difference of two consecutive `brillouin`⋱ `ZonePolygonsSC`) zones have the same volume.

```
In[119]:= Table[volume[brillouinZonePolygonsSC[i]],
              {i, Length[DownValues[brillouinZonePolygonsSC]]}]

Out[119]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
           13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}
```

For a nice picture of the Brillouin zones, we do not want just to display the individual polygons calculated. It would show edges inside plane polygons induced by the planes bounding the symmetry unit. To avoid this, we will display all polygons with the `EdgeForm[]` directive and instead calculate all edges. To do this, we test if two neighboring polygons are in the same plane (this means if their normals are parallel). This must also be done across the boundaries of the unit under consideration. The functions `boundary`*i*`EdgeQ` determine if a given edge is on the unit boundary.

```
In[120]:= boundary1EdgeQ[Edge[{{p1x_, p1y_, p1z_}, {p2x_, p2y_, p2z_}}]] :=
                                                 p1y === p2y === 0;
          boundary2EdgeQ[Edge[{{p1x_, p1y_, p1z_}, {p2x_, p2y_, p2z_}}]] :=
                                            p1x === p1y && p2x === p2y;
          boundary3EdgeQ[Edge[{{p1x_, p1y_, p1z_}, {p2x_, p2y_, p2z_}}]] :=
                                            p1x === p1z && p2x === p2z;
```

inTheSamePlaneQ determines if the two polygons are in the same plane.

```
In[123]:= inTheSamePlaneQ[{Polygon[l1_], Polygon[l2_]}] :=
          With[{normal1 = Cross[l1⟦1⟧ - l1⟦2⟧, l1⟦1⟧ - l1⟦3⟧],
                normal2 = Cross[l2⟦1⟧ - l2⟦2⟧, l2⟦1⟧ - l2⟦3⟧]},
               (normal1.normal2)^2 === normal1.normal1 normal2.normal2]
```

The function visibleEdges finally generates all edges between polygons that do not belong into one plane and returns the visible edges as explicit line primitives.

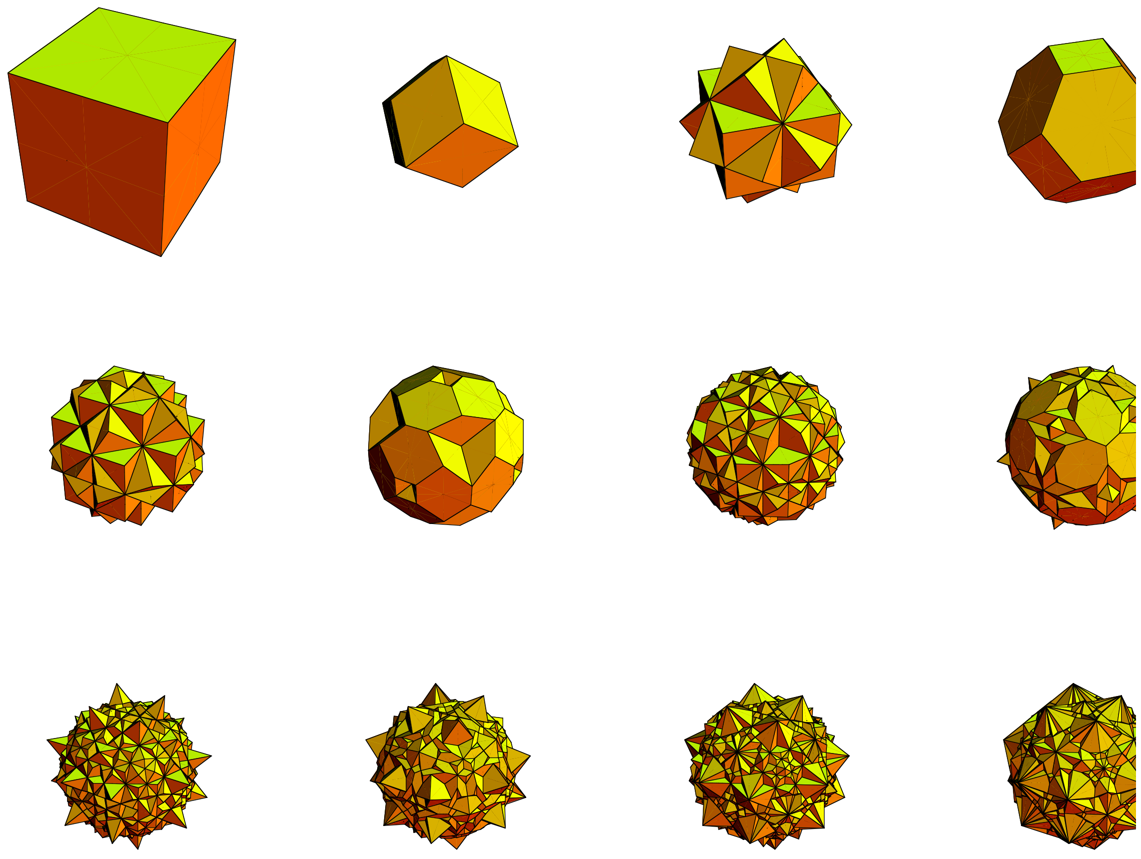
```
In[124]:= visibleEdges[unitPolys_] :=
          Module[{unitPolyEdges, allEdgesV, boundary1Polys,
                  boundary2Polys, boundary3Polys, allPolys},
           (* all edges of the polygons *)
          unitPolyEdges = {#, edges[#]}& /@ unitPolys;
          allEdgesV = Union[Flatten[Last /@ unitPolyEdges]];
          (* the neighboring polygons *)
          {boundary1Polys, boundary2Polys, boundary3Polys} =
              Function[f, First /@
                  Select[unitPolyEdges, MemberQ[#⟦2⟧, _?f]&]] /@
                        {boundary1EdgeQ, boundary2EdgeQ, boundary3EdgeQ};
          (* add mirrored polygons along the edges of the unit *)
          augmentedPolys =
          {Apply[{#1, -#2, #3}&, boundary1Polys, {3}],
           Apply[{#2,  #1, #3}&, boundary2Polys, {3}],
           Apply[{#3,  #2, #1}&, boundary3Polys, {3}]};
          allPolys = Flatten[{unitPolys, augmentedPolys}];
          (* build polygonsFromEdgeV function *)
          Clear[polygonsFromEdgeV];
          (polygonsFromEdgeV[#] = {})& /@ allEdgesV;
          Function[p, Map[(polygonsFromEdgeV[#] = {polygonsFromEdgeV[#], p})&,
                 Intersection[edges[p], allEdgesV], {1}]] /@ allPolys;
          (polygonsFromEdgeV[#] = Flatten[polygonsFromEdgeV[#]])& /@ allEdgesV;
           If[inTheSamePlaneQ[polygonsFromEdgeV[#]], {}, Line @@ #]& /@ allEdgesV]
```

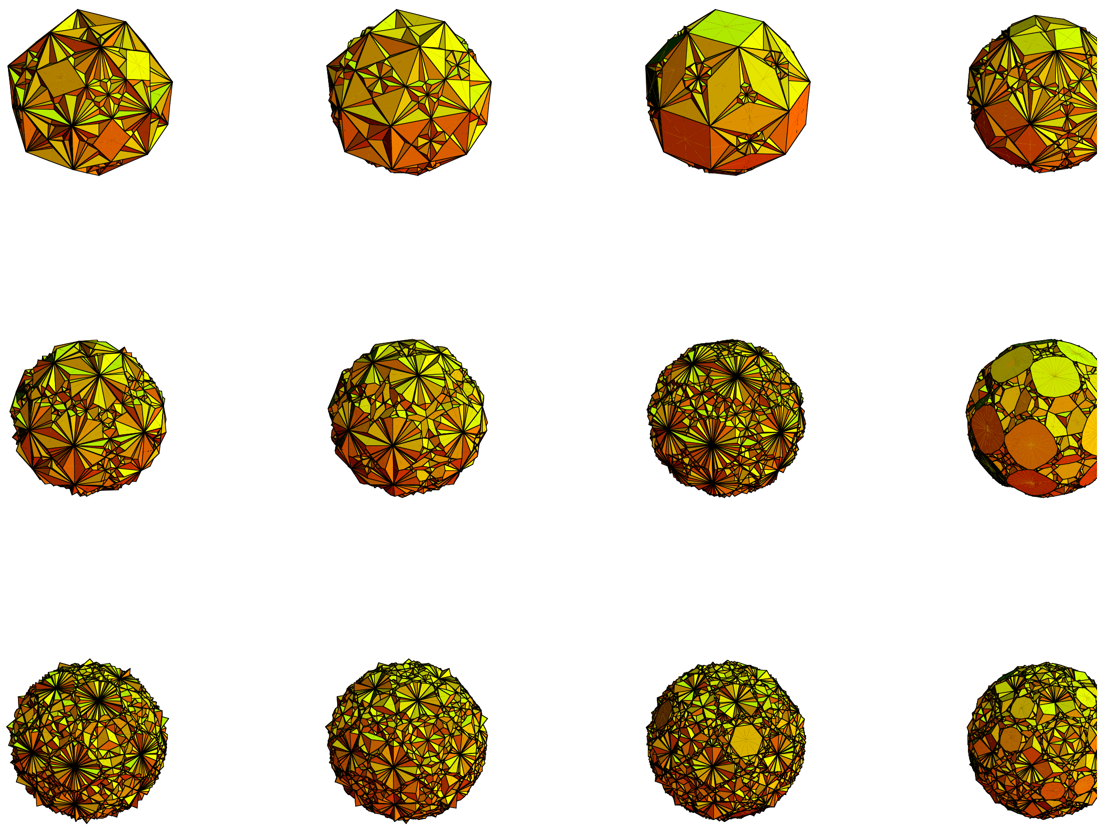
Until now, we have only the polygons and edges inside the symmetry unit. All other 47 equivalent sets of polygons and edges are generated by the function makeAllCube. It recursively rotates and mirrors the polygons and edges from the unit.

```
In[125]:= makeAllCube[(pl:Polygon | Line)[l_]] :=
          pl /@ Join[#, Apply[{#2, #3, #1}&, #, {2}],
                        Apply[{#3, #1, #2}&, #, {2}]]&[
                         Join[#, Apply[{#2, #1, -#3}&, #, {2}]]&[
                          Join[#, Apply[{#2, #1, #3}&, #, {2}]]&[
                           Join[#, Apply[{-#1, #2, #3}&, #, {2}]]&[
                                {l, Apply[{#1, -#2, #3}&, l, {1}]}]]]]
```

Now, finally, comes the moment the reader has been waiting for a long time: the actual pictures of the Brillouin zones.

```
In[126]:= showBrillouinZone[unitPolys_, color_, opts___] :=
          Module[{unitEdges, allPolys, allEdges},
                 (* the visible edges inside the unit *)
                 unitEdges = visibleEdges[unitPolys];
                  (* make all polygons in all units *)
                 allPolys = makeAllCube /@ unitPolys;
                  (* make all edges in all units *)
                 allEdges = makeAllCube /@ Flatten[unitEdges];
                  (* show polygons and edges *)
                 Show[Graphics3D[
                  {EdgeForm[], Thickness[0.001], color, allPolys, allEdges}],
                   opts, Boxed → False, PlotRange → All, SphericalRegion → True]]

In[127]:= Do[Show[GraphicsArray[
           Table[showBrillouinZone[brillouinZonePolygonsSC[4 k + l],
                                  SurfaceColor[Hue[0.06], Hue[0.32], 2.1],
                                  DisplayFunction -> Identity],
                 {l, 4}]]], {k, 0, 5}]
```

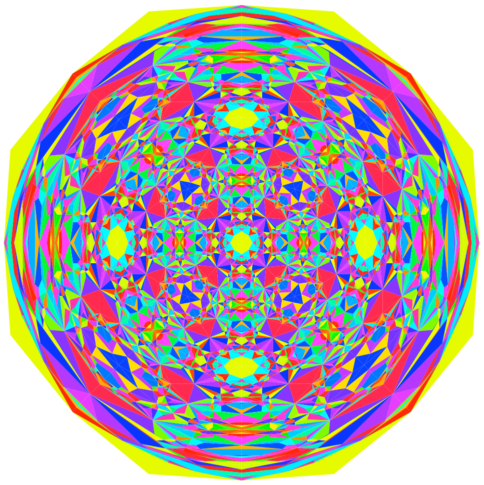




To see the complicated face structure of, say, the 25th zone more clearly we unfold the spherical polyhedron into a circle.

```
In[128]:= project[{x_, y_, z_}] :=
          Module[{φ = ArcTan[x, y], ϑ = ArcCos[z/Sqrt[x^2 + y^2 + z^2]]},
                  ϑ {Cos[φ], Sin[φ]}]
```

```
In[129]:= Show[Graphics[{Hue[Random[]], #}& /@ Partition[Map[project,
          With[{mp = Plus @@ #[[1]]/Length[#[[1]]]},
               Polygon[(mp + (1. - 10^-10)(# - mp))& /@ #[[1]]]]& /@
             Cases[showBrillouinZone[brillouinZonePolygonsSC[25], Null,
               DisplayFunction -> Identity], _Polygon, Infinity], {-2}],
                 48]], AspectRatio -> Automatic];
```



To get a better view on the Brillouin zone as a whole, we make a “solid wireframe” based on the visible edges. The function `showSolidWireFrameBrillouinZone` is implementing this construction.

```
In[130]:= showSolidWireFrameBrillouinZone[unitPolys_, color_,
             inplaneContractionFactor_, radialContractionFactor_, opts___] :=
          Module[{unitEdges, edgePolygonPairs, allPolys},
                 (* the visible edges inside the unit *)
                 unitEdges = visibleEdges[unitPolys];
                 (* the edges of the polygons to be beamized *)
                 edgePolygonPairs = Function[edge,
                    {edge, Select[unitPolys, MemberQ[edges[#],
                                  Edge @@ edge]&]}] /@ Flatten[unitEdges];
                 (* make all beams *) allPolys = makeAllCube /@
                   (Flatten[makeBeam[#, radialContractionFactor,
                                        inplaneContractionFactor,
                      Flatten[unitEdges]]& /@ edgePolygonPairs]);
                 (* show polygons and edges *)
                Show[Graphics3D[{EdgeForm[], Thickness[0.001],
                                 color, allPolys}], opts, Boxed → False,
                     PlotRange → All, SphericalRegion → True]]
```

For a single edge of a given polygon, the function `makeBeam` generates the explicit polygons that form the wireframe. To avoid intersecting polygons, we have to differentiate between edges that have a point with another edge in common and edges that do not.

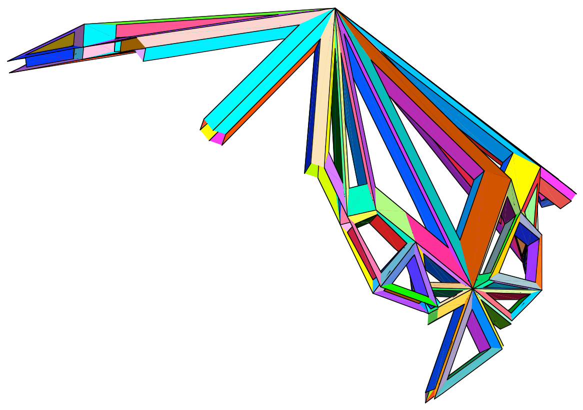
```
In[131]:= makeBeam[{edge_, polys_List}, f1_, f2_, allVisisbleEdges_] :=
          makeBeam[edge, #, f1, f2, allVisisbleEdges]& /@ polys;

          makeBeam[Line[edge_], Polygon[l_], f1_, f2_, allVisisbleEdges_] :=
          Module[{pos1 = Position[l, edge[[1]]][[1, 1]],
                  pos2 = Position[l, edge[[2]]][[1, 1]],
                  neighbor1, neighbor2, neighborEdge1,
                  neighborEdge2, innerPoints, edgeContracted,
                  innerPointsContracted},
          (* the neighboring point of the edge under consideration *)
          neighbor1 = Complement[Which[pos1 === 1, {l[[-1]], l[[2]]},
                                       pos1 === Length[l], {l[[-2]], l[[1]]},
                                       True, {l[[pos1 - 1]], l[[pos1 + 1]]}],
                                 {edge[[2]]}][[1]];
          neighbor2 = Complement[Which[pos2 === 1, {l[[-1]], l[[2]]},
                                       pos2 === Length[l], {l[[-2]], l[[1]]},
                                       True, {l[[pos2 - 1]], l[[pos2 + 1]]}],
                                 {edge[[1]]}][[1]];
          (* the two neighboring edges *)
          neighborEdge1 = Line[Sort[{edge[[1]], neighbor1}]];
          neighborEdge2 = Line[Sort[{edge[[2]], neighbor2}]];
          (* inner points inside the polygon *)
          innerPoints =
          {If[MemberQ[allVisisbleEdges, neighborEdge1],
              edge[[1]] + (1 - f1)(neighbor1 + edge[[2]] - 2edge[[1]]),
              neighbor1 + f1(edge[[1]] - neighbor1)],
           If[MemberQ[allVisisbleEdges, neighborEdge2],
              edge[[2]] + (1 - f1)(neighbor2 + edge[[1]] - 2edge[[2]]),
              neighbor2 + f1(edge[[2]] - neighbor2)]};
          (* radially contracted edge *)
          edgeContracted = Map[f2 #&, edge];
          {neighbor1Contracted, neighbor2Contracted} =
                                 f2 {neighbor1, neighbor2};
          (* inner points of the radially contracted polygon *)
          innerPointsContracted =
          {If[MemberQ[allVisisbleEdges, neighborEdge1],
              edgeContracted[[1]] + (1 - f1)*
              (neighbor1Contracted + edgeContracted[[2]] - 2 edgeContracted[[1]]),
              neighbor1Contracted + f1*
              (edgeContracted[[1]] - neighbor1Contracted)],
           If[MemberQ[allVisisbleEdges, neighborEdge2],
              edgeContracted[[2]] + (1 - f1)*
              (neighbor2Contracted + edgeContracted[[1]] - 2 edgeContracted[[2]]),
              neighbor2Contracted + f1(edgeContracted[[2]] -
                                       neighbor2Contracted)]};
          (* polygons and lines forming the beams *)
          {Polygon[Join[edge, Reverse[innerPoints]]],
           Polygon[Join[edgeContracted, Reverse[innerPointsContracted]]],
           Polygon[Join[innerPoints, Reverse[innerPointsContracted]]],
           Line[edge], Line[innerPoints],
           Line[edgeContracted], Line[innerPointsContracted],
           Line[{innerPoints[[1]], innerPointsContracted[[1]]}],
           Line[{innerPoints[[2]], innerPointsContracted[[2]]}]}]
```

To better see the functionality of `makeBeams` (which generates only the absolutely necessary polygons), we do not display a whole Brillouin zone, but rather only the parts from a unit. But `showSolidWireFrameBrillouinZone` was implemented to display the whole Brillouin zone. Fortunately, it is possible to manipulate programs easily in *Mathematica*, so that we just take the existing definition of `showSolidWireFrameBrillouinZone` and modify it not to generate all other unit parts.

```
In[133]:= DownValues[showSolidWireFrameBrillouinZoneInUnit] =
              DownValues[showSolidWireFrameBrillouinZone] /.
                {showSolidWireFrameBrillouinZone →
                  showSolidWireFrameBrillouinZoneInUnit, makeAllCube → Identity};
```

The following picture shows the polygons of the beams inside the unit of the 17th Brillouin zone. For easier discrimination of the polygons, we color them randomly.

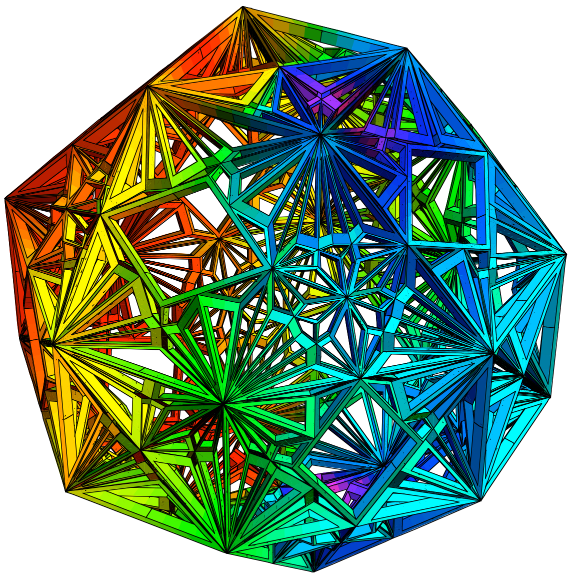
```
In[134]:= Show[showSolidWireFrameBrillouinZoneInUnit[
            brillouinZonePolygonsSC[17], {}, 0.98, 0.88,
            PlotRange → All, DisplayFunction → Identity] /.
            p_Polygon :→ {SurfaceColor[Hue[Random[]], Hue[Random[]], 3Random[]], p},
            DisplayFunction → $DisplayFunction];
```



Here is the 13th Brillouin zone shown in the solid wireframe manner. This time, we color every polygon individually. To get a smooth color variation, we subdivide the potentially thin and long edge polygons that form the beams.

```
In[135]:= subdividePolygon[Polygon[{p1_, p2_, p3_, p4_}], lMax_] :=
          Module[{n},
                 Apply[Polygon[Join[#1, Reverse[#2]]]&,
                      Transpose[Partition[#, 2, 1]& /@
                 If[(* which edge is the longest one? *)
                    (p1 - p2).(p1 - p2) < (p2 - p3).(p2 - p3),
                     n = Ceiling[Sqrt[(p2 - p3).(p2 - p3)]/lMax];
                    {Table[p1 + i/n(p4 - p1), {i, 0, n}],
                     Table[p2 + i/n(p3 - p2), {i, 0, n}]},
                     n = Ceiling[Sqrt[(p1 - p2).(p1 - p2)] /lMax];
                    {Table[p1 + i/n(p2 - p1), {i, 0, n}],
                     Table[p4 + i/n(p3 - p4), {i, 0, n}]}]], {1}]]
```

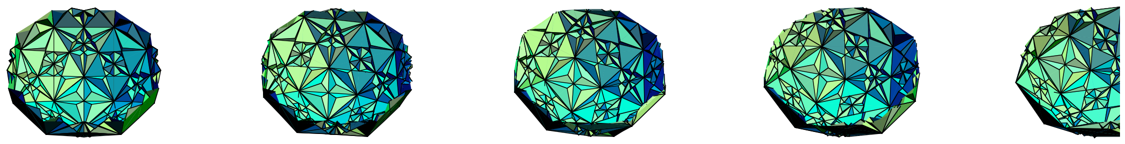
```
In[136]:= Show[showSolidWireFrameBrillouinZone[
              brillouinZonePolygonsSC[13], {}, 0.96, 0.9,
                   PlotRange → All, DisplayFunction → Identity] /.
                   p_Polygon :→ subdividePolygon[p, 0.1] /.
                   (* rainbow coloring in a spherical coordinate system *)
                   Polygon[l_] :→
                   {Module[{xmp, ymp, zmp, col, φ, ϑ},
                           {xmp, ymp, zmp} = Plus @@ l/4;
                            φ = ArcTan[xmp, ymp];
                            ϑ = ArcCos[zmp/({xmp, ymp, zmp}.{xmp, ymp, zmp})];
                            col = Hue[(1/2 + 1/2 Cos[ϑ]^2) Cos[φ/2]^2];
                           SurfaceColor[col, col, 2.6]], Polygon[l]},
               DisplayFunction → $DisplayFunction];
```



The Brillouin zones are not the “whole” polyhedra shown above, but rather the thin objects between two successive polyhedra. To get a better impression about them, we take two of them and slice them with the $z = 0$-plane.

```
In[137]:= hollowBrillouinZone =
          {{makeAllCube /@ brillouinZonePolygonsSC[14],
            makeAllCube /@ Flatten[visibleEdges[brillouinZonePolygonsSC[14]]]},
            Map[(1 + 10^-6)#&,
            {makeAllCube /@ brillouinZonePolygonsSC[15],
             makeAllCube /@ Flatten[visibleEdges[
               brillouinZonePolygonsSC[15]]]}, {-2}]};
```

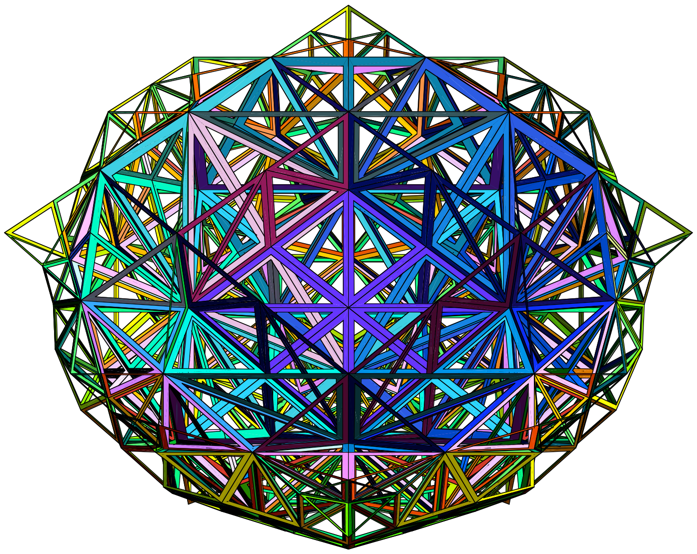
```
In[138]:= Show[GraphicsArray[
          Table[(* the rotation matrix *)
             ℛ = N[{{1, 0, 0}, {0, Cos[φ], Sin[φ]}, {0, -Sin[φ], Cos[φ]}}];
              Show[Graphics3D[{EdgeForm[], Thickness[0.001],
                               SurfaceColor[Hue[0.48], Hue[0.21], 2.1],
                     Sequence @@ Flatten[Map[ℛ.#&,
                        hollowBrillouinZone, {-2}]]}],
                   Boxed → False, PlotRange → {All, All, {-2, 0}},
                   SphericalRegion → True, ViewPoint → {1, 1, 2},
                   DisplayFunction → Identity], {φ, 0, π/4, Pi/4/4}]]];
```



Another possibility to visualize their relative orientation is to stack some of them into each other and make holes in all of their polygons to look inside.

```
In[139]:= makeHole[Polygon[l_], factor_] :=
          Module[{mp = Plus @@ l/Length[l], innerPoints},
                 innerPoints = (mp + factor(# - mp))& /@ l;
                 (* form new polygons *)
                {MapThread[Polygon[Join[#1, Reverse[#2]]]&,
                 {Partition[Append[#, First[#]]&[l], 2, 1],
                  Partition[Append[#, First[#]]&[innerPoints], 2, 1]}],
                Line[Append[#, First[#]]]&[innerPoints],
                Line[Append[#, First[#]]]&[l]}]
```

```
In[140]:= Show[Graphics3D[{EdgeForm[], Thickness[0.0001],
            Table[{SurfaceColor[Hue[Random[]], Hue[Random[]], 3Random[]],
                  makeHole[#, 0.8]& /@ Flatten[makeAllCube /@
                                        brillouinZonePolygonsSC[i]],
                  makeAllCube /@ Flatten[visibleEdges[
                        brillouinZonePolygonsSC[i]]]}, {i, 8}]}],
              Boxed → False, PlotRange → {All, All, {-2, 0}},
              SphericalRegion → True, ViewPoint → {1, 1, 2}];
```



We could now go on and color all faces in one color. The function `groupPolygonsIntoFaces` groups the individual polygons into the list of polygons that form a face inside the unit.

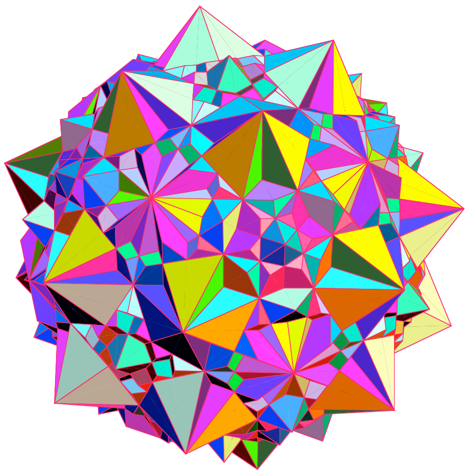
```
In[141]:= groupPolygonsIntoFaces[l_] :=
          Module[{stillToBeUsedPolygons, coloredPolygonList,
                  subList, actualPoly, neighbors, parallelneighbors},
          stillToBeUsedPolygons = l;
          coloredPolygonList = {};
          While[stillToBeUsedPolygons =!= {},
                subList = {stillToBeUsedPolygons[[1]]};
                stillToBeUsedPolygons = Rest[stillToBeUsedPolygons];
                While[(* select all neighbor polygons *)
                      parallelneighbors = Union[Flatten[
                           Table[actualPoly = subList[[i]];
                      neighbors = Select[stillToBeUsedPolygons,
                                        (Intersection[edges[actualPoly],
                                                      edges[#]] =!= {})&];
                      (* select parallel neighbor polygons *)
                       Select[neighbors, inTheSamePlaneQ[{actualPoly, #}]&],
                                 {i, Length[subList]}]]]];
                      parallelneighbors =!= {} && stillToBeUsedPolygons =!= {},
                      (* update lists *)
                      subList = Flatten[{subList, parallelneighbors}];
                      stillToBeUsedPolygons = DeleteCases[stillToBeUsedPolygons,
                           Alternatives @@ parallelneighbors]];
                         AppendTo[coloredPolygonList, subList]];
           (* return list of lists with parallel polygons *)
              coloredPolygonList]
```

The corresponding function `showColoredBrillouinZone` generates a graphic with the colored faces.

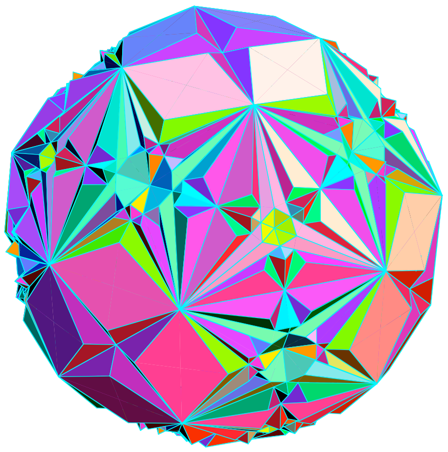
```
In[142]:= showColoredBrillouinZone[unitPolys_, opts___] :=
          Module[{unitEdges, allPolys, allEdges},
                  (* the visible edges inside the unit *)
                  unitEdges = visibleEdges[unitPolys];
                  (* make all polygons in all units *)
                  allPolys = Map[makeAllCube,
                                 groupPolygonsIntoFaces[unitPolys], {2}];
                     (* make all edges in all units *)
                  allEdges = makeAllCube /@ Flatten[unitEdges];
                  (* show polygons and edges *)
                  Show[Graphics3D[{EdgeForm[], Thickness[0.001], Hue[Random[]],
                                  {SurfaceColor[Hue[Random[]],  Hue[Random[]],
                                       3Random[]], #}& /@ allPolys, allEdges}],
                       opts, Boxed → False, PlotRange → All, SphericalRegion → True]]
```

Here are three examples.

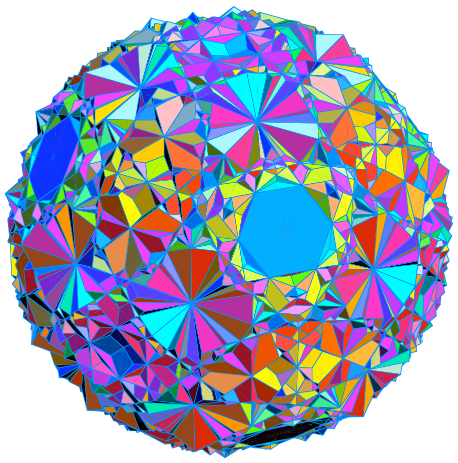
```
In[143]:= showColoredBrillouinZone[brillouinZonePolygonsSC[11]];
```



```
In[144]:= showColoredBrillouinZone[brillouinZonePolygonsSC[16]];
```



```
In[145]:= showColoredBrillouinZone[brillouinZonePolygonsSC[23]];
```



Enough playing with making pictures of Brillouin zones of a simple 3D cubic lattice; it is time go on to other lattices.

5. Brillouin Zones of a Body-Centered Lattice

Now that we have implemented all functions needed we just change the lattice point that defines the lattice and reevaluate the various functions needed to calculate the Brillouin zones.

We start by generating the lattice points. Because the reciprocal lattice of a body-centered lattice is a face-centered one, we have to use a face-centered lattice in the direct space now.

```
In[146]:= latticePointListBC[δ_] :=
          With[{n = Ceiling[δ]},
          Select[Union[Join[DeleteCases[
          Flatten[Table[{i, j, k}, {i, -n, n}, {j, -n, n}, {k, -n, n}], 2],
                                       {0, 0, 0}],
                   (* the points in the centers of the faces *)
          Flatten[Table[{i, j, k} + {1, 1, 0}/2,
                            {i, -n, n}, {j, -n, n}, {k, -n, n}], 2],
          Flatten[Table[{i, j, k} + {1, 0, 1}/2,
                            {i, -n, n}, {j, -n, n}, {k, -n, n}], 2],
          Flatten[Table[{i, j, k} + {0, 1, 1}/2,
                            {i, -n, n}, {j, -n, n}, {k, -n, n}], 2]]], #.# <= δ^2&]]

In[147]:= maxDist = 3;
          Length[latticePoints = latticePointListBC[maxDist]]

Out[148]=  458
```

We form all planes.

```
In[149]:= planes = Join[toPlane /@ latticePoints, symmetrySlicingPlanes];
```

Because we stored the tessellations of the planes with `tessellatePlane`, we remove all but the general definition.

```
In[150]:= DownValues[tessellatePlane] = DownValues[tessellatePlane][[-1]];
```

We calculate all polygons in our unit.

```
In[151]:= polygonsInUnit = Flatten[Select[
            tessellatePlane1[#, maxDist/2], inUnitQ]& /@ latticePoints];
```

We single out the starting polygons on the z-axis.

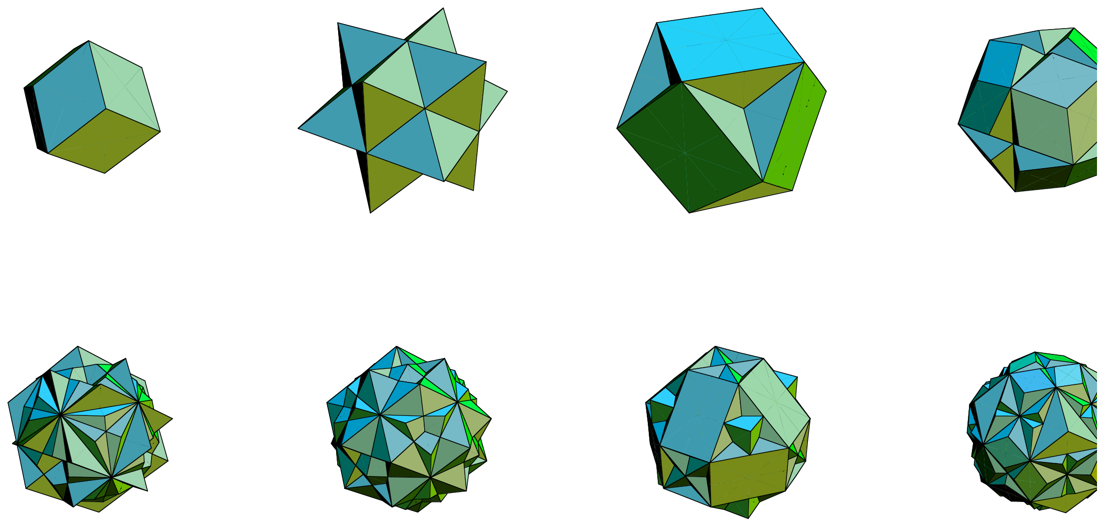
```
In[152]:= sortedStartingPolygons =
            sortPolygonsOnZAxis[selectPolygonsOnZAxis[polygonsInUnit]];

In[153]:= Length[sortedStartingPolygons]

Out[153]= 38
```

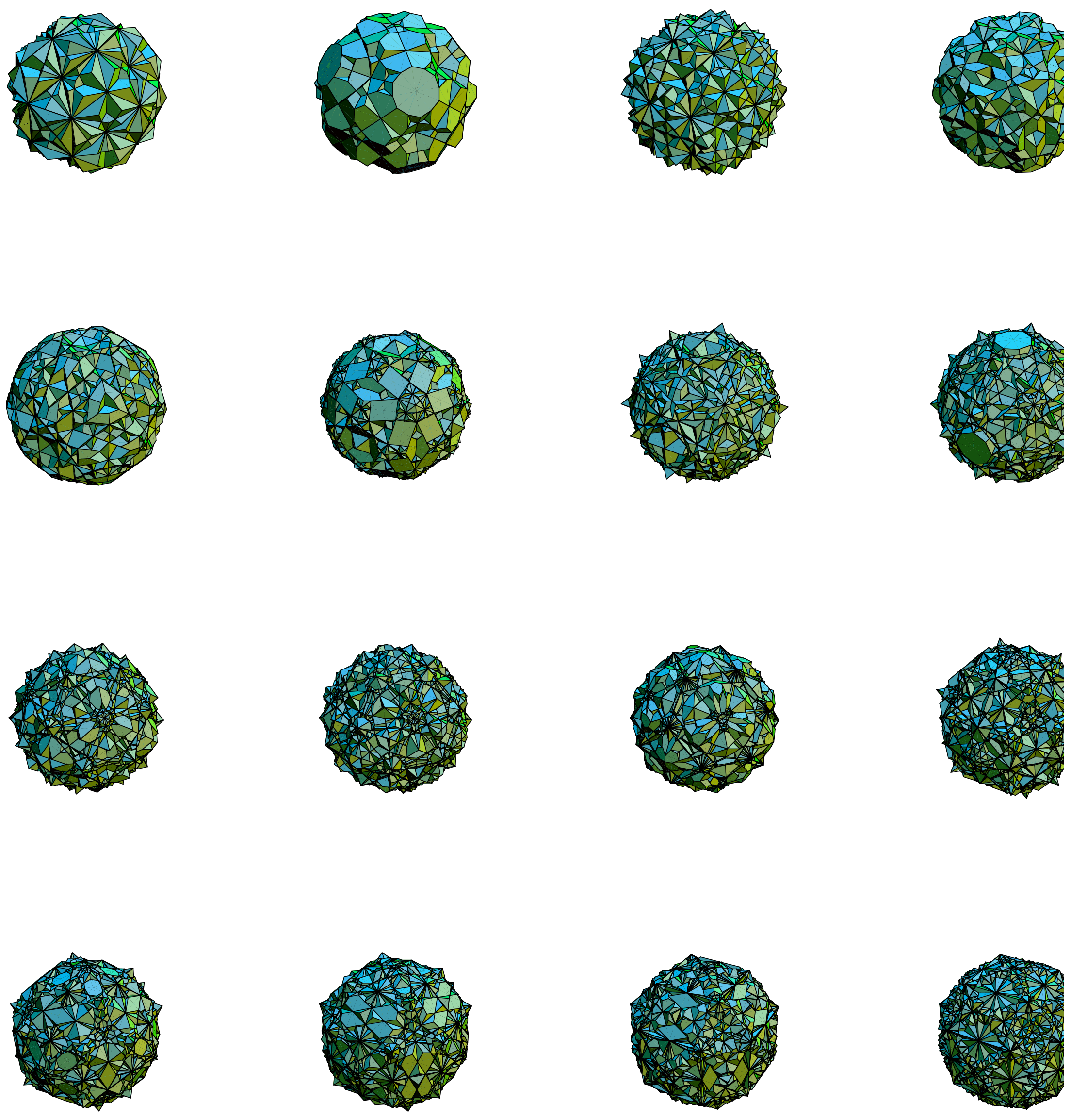
We calculate the Brillouin zones.

```
In[154]:= edgesAndPolygons[polygonsInUnit];

In[155]:= Do[startPoly = sortedStartingPolygons⟦i⟧;
             freeStartEdges = Complement[#,
                     Select[#, outerEdgeQ]]&[edges[startPoly]];
             upDatePolygonsFromEdgeDefinitions[startPoly];
             brillouinZonePolygonsBC[i] =
                  buildBrillouinZone[{{startPoly, freeStartEdges}}],
            {i, 25}]
```

And finally we visualize them.

```
In[156]:= Do[Show[GraphicsArray[
             Table[showBrillouinZone[brillouinZonePolygonsBC[4 k + l],
                                     SurfaceColor[Hue[0.32], Hue[0.76], 1.9],
                                     DisplayFunction -> Identity],
                  {l, 4}]]], {k, 0, 5}]
```





Let us again check their volumes—now that we have $6\times\frac{1}{2}+8\times\frac{1}{8}=4$ lattice points in each cell, the volume is 1/4.

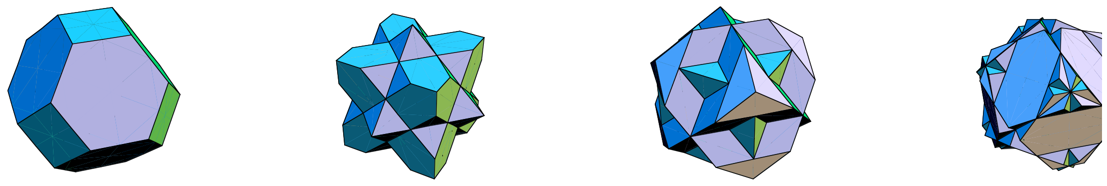
```
In[157]:= Table[volume[brillouinZonePolygonsBC[i]], {i, 25}]
```

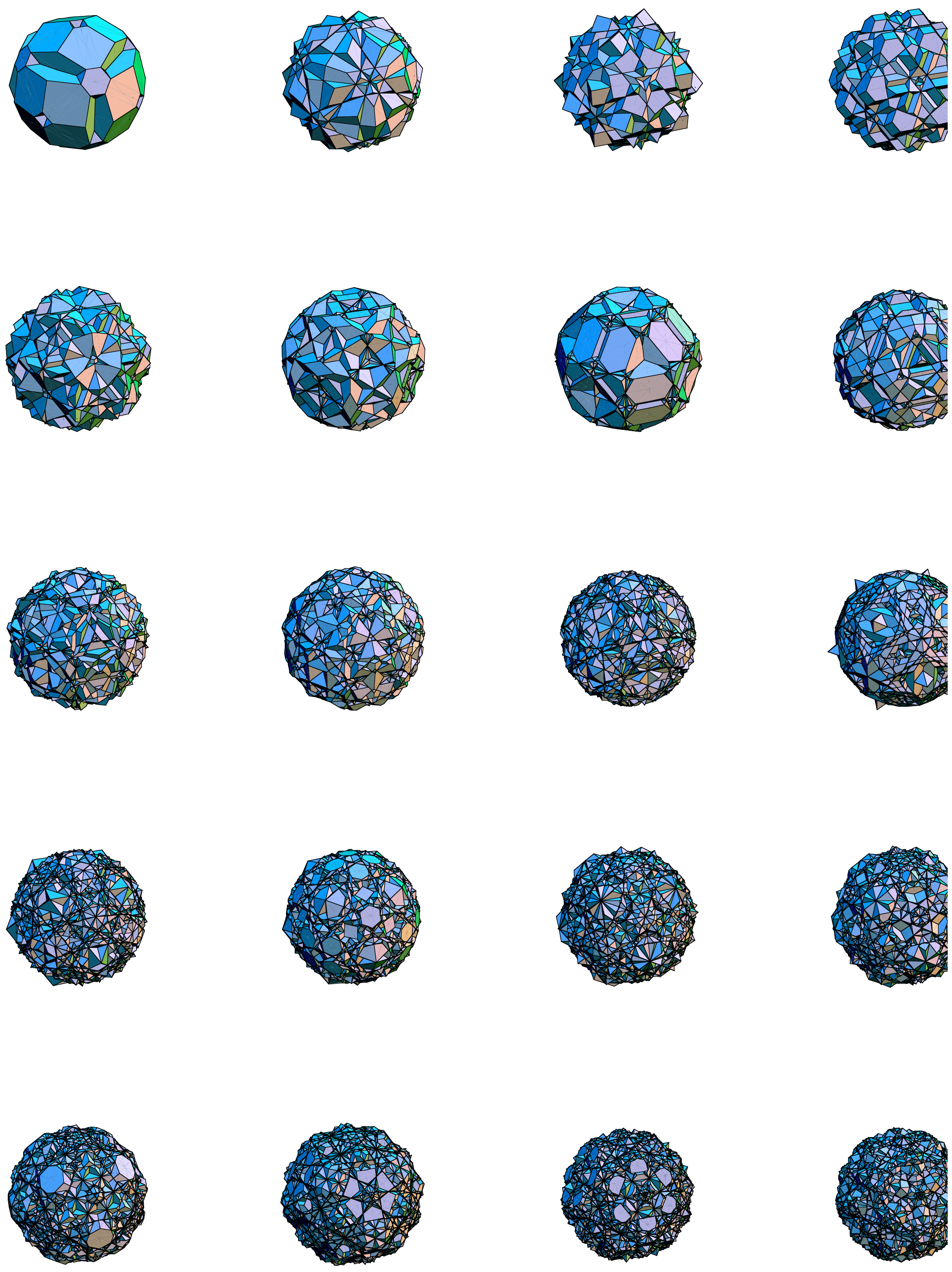
Out[157]=
$$\left\{\frac{1}{4},\frac{1}{2},\frac{3}{4},1,\frac{5}{4},\frac{3}{2},\frac{7}{4},2,\frac{9}{4},\frac{5}{2},\frac{11}{4},3,\frac{13}{4},\frac{7}{2},\frac{15}{4},4,\frac{17}{4},\frac{9}{2},\frac{19}{4},5,\frac{21}{4},\frac{11}{2},\frac{23}{4},6,\frac{25}{4}\right\}$$

6. Brillouin Zones of a Face-Centered Lattice

Not much explanation is needed now that it is the third time we use the code.

```
In[158]:= (* the lattice points to take into account *)
          latticePointListFC[δ_] :=
          With[{n = Ceiling[δ]},
               Select[Union[Join[DeleteCases[
               Flatten[Table[{i, j, k}, {i, -n, n}, {j, -n, n}, {k, -n, n}], 2],
                                              {0, 0, 0}],
             (* the points in the center of the cubes *)
               Flatten[Table[{i, j, k} + {1, 1, 1}/2,
                             {i, -n, n}, {j, -n, n}, {k, -n, n}], 2]]],
                      #.# <= δ^2&]]

In[160]:= maxDist = 7/2;
          Length[latticePoints = latticePointListFC[maxDist]]

Out[161]=  338

In[162]:= planes = Join[toPlane /@ latticePoints, symmetrySlicingPlanes];

In[163]:= DownValues[tessellatePlane] = DownValues[tessellatePlane]⟦-1⟧;

In[164]:= polygonsInUnit = Flatten[Select[
           tessellatePlane1[#, maxDist/2], inUnitQ]& /@ latticePoints];

In[165]:= sortedStartingPolygons =
            sortPolygonsOnZAxis[selectPolygonsOnZAxis[polygonsInUnit]];

In[166]:= edgesAndPolygons[polygonsInUnit];

In[167]:= Do[startPoly = sortedStartingPolygons⟦i⟧;
             freeStartEdges = Complement[#,
                    Select[#, outerEdgeQ]]&[edges[startPoly]];
             upDatePolygonsFromEdgeDefinitions[startPoly];
             brillouinZonePolygonsFC[i] =
                   buildBrillouinZone[{{startPoly, freeStartEdges}}],
             {i, 25}]

In[168]:= Do[Show[GraphicsArray[
             Table[showBrillouinZone[brillouinZonePolygonsFC[4 k + l],
                                     SurfaceColor[Hue[0.45], Hue[0.87], 2.2],
                                     DisplayFunction -> Identity],
                   {l, 4}]]], {k, 0, 5}]
```





Let us us again check their volumes—now that we have 8×1/8+1×1=2 lattice points in each cell, the volume is 1/2.

```
In[169]:= Table[volume[brillouinZonePolygonsFC[i]], {i, 25}]
```

$$\left\{\frac{1}{2}, 1, \frac{3}{2}, 2, \frac{5}{2}, 3, \frac{7}{2}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, \frac{17}{2}, 9, \frac{19}{2}, 10, \frac{21}{2}, 11, \frac{23}{2}, 12, \frac{25}{2}\right\}$$

This ends our journey through the world of Brillouin zones. The (interested) reader now can go on and can calculate and visualize still higher order zones or extend the construction to hexagonal lattices or add Fermi spheres [540✱] or so on.

References

✱9 E. L. Allgower, P. H. Schmidt. *Math. Comput.* 46, 171 (1986).

✱26 N. W. Ashcroft, N. D. Mermin. *Solid State Physics*, Saunders, Philadelphia, 1976.

✱31 F. Aurenhammer. *ACM Comput. Surv.* 23, 345 (1991).

✱49 L. Bieberbach. *Monatshefte Math. Phys.* 48, 509 (1938).

✱60 L. Brillouin. *Wave Propagation in Periodic Structures*, McGraw-Hill, New York 1946.

✱97 A. P. Crackwell, K. C. Wong. *The Fermi Surface*, Clarendon Press, Oxford, 1973.

✱132 H. Edelsbrunner. *Algorithms in Combinatorical Geometry*, Springer-Verlag, Berlin, 1987.

✱196 A. B. Goncharov. *Quantum* 9, 4 (1998).

✱258 J. D. Joannopoulos, R. D. Meade, J. N. Winn. *Photonic Crystals*, Princeton University Press, Princeton, 1995.

✱262 G. A. Jones. *Bull. Lond. Math. Soc.*16, 241 (1984).

✱276 C. Kittel. *Introduction to Solid State Physics*, Wiley, New York, 1986.

✱301 P. T. Landsberg in P. T. Landsberg (ed.). *Solid State Theory*, Wiley, London, 1969.

✱338 L. Michel. *Phys. Rep.* 341, 265 (2001).

✱364 A. Okabe, B. Boots, K. Sugihara. *Spatial Tessellations—Concepts and Applications of Voronoi Diagrams*, Wiley, Chichester, 1995.

✱457 M. M. Skriganov. *Zap. Nautn. Sem. Leningrad. Otdel. Mat. Inst. Steklov.* 134, 206 (1984).

✱488 M. Trott. *The Mathematica GuideBook for Programming*, Springer-Verlag, New York, 2003.

✱490 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2003.

✱495 J. J. P Veerman, M. M. Peixoto, A. C. Rocha, S. Sutherland. *arXiv:math.MG*/9806154 (1998). *Get Preprint*

✱539 J. M. Ziman. *Principles of the Theory of Solids*, Cambridge University Press, Cambridge, 1986.

✱540 Zimbovskaya. *Local Geometry of the Fermi Surface*, Springer-Verlag, New York, 2001.