Algorithms for the Generation of Generalized Monkhorst-Pack Grids

Yunzhe Wang¹, Pandu Wisesa¹, Adarsh Balasubramanian¹, and Tim Mueller¹,*

¹ Department of Materials Science and Engineering, Johns Hopkins University, Baltimore, Maryland 21218, USA

E-mail: tmueller@jhu.edu

Abstract

Computational simulations of crystalline materials properties have become an increasingly important research paradigm over the last decade. A routine operation in such calculations is the evaluation of integrals over the Brillouin zone. Traditional Monkhorst-Pack grids are widely used across popular simulation packages. Researchers have previously demonstrated that generalised Monkhorst-Pack k-point grids are much more efficient than traditional Monkhorst-Pack grids and can roughly double the speed of well-converged calculations on crystalline materials. In this paper, we present algorithms that can rapidly generate the optimized generalized Monkhorst Pack grid both dynamically and using a pre-generated database. The algorithms identify highly efficient grids by performing an exhaustive search over all symmetry-preserving grids, and they also incorporate designs to maximally accelerate such searches. Implementations of these algorithms in three different formats are also provided, to meet the diverse demands of users.

Keywords: Brillouin zone, k-points
1. Introduction

Computational materials research has become increasingly vital in probing the properties of crystalline materials, especially in screening materials at a large scale to accelerate material discoveries for a wide range of applications. A routine operation for calculating the properties of crystalline materials is the evaluation of integrations over the first Brillouin zone, which can be approximated by discretely sampling the Brillouin Zone at a set of points known as $k$-points. For most methods, the computational cost of a calculation scales linearly with the number of the symmetrically irreducible $k$-points used to approximate the integral.

Many popular computational materials software packages generate $k$-points using the Monkhorst-Pack scheme [1], which creates $k$-point grids that are regular and aligned with the reciprocal lattice vectors. In an $m_1 \times m_2 \times m_3$ Monkhorst-Pack grid, the coordinates of the $k$-points are given by

$$
k = \frac{n_1}{m_1} \mathbf{b}_1 + \frac{n_2}{m_2} \mathbf{b}_2 + \frac{n_3}{m_3} \mathbf{b}_3 + \mathbf{s}, \quad n_1 = 0 \ldots m_1 - 1, \ n_2 = 0 \ldots m_2 - 1, \ n_3 = 0 \ldots m_3 - 1 \quad (1)$$

where $m_1$, $m_2$, and $m_3$ are positive integers, $\mathbf{b}_1$, $\mathbf{b}_2$ and $\mathbf{b}_3$ are reciprocal lattice vectors, and $\mathbf{s}$ represents a shift vector that moves the grid away from the origin (known as the $\Gamma$ point in reciprocal space). There exists a mapping between each regular $k$-point grid and a real-space superlattice that defines the Born-von Karman boundary conditions for the periodicity of the wave functions [2, 3]. The superlattice corresponding to the $k$-point grid defined by equation (1) is given by

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}^T = \mathbf{M} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}^T \quad (2)$$

where $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$ represent the real-space primitive lattice vectors, $\mathbf{c}_1$, $\mathbf{c}_2$, and $\mathbf{c}_3$ represent the lattice vectors of the superlattice, and the transformation matrix $\mathbf{M}$ is equal to

$$\mathbf{M} = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} \quad (3)$$

The reciprocal primitive lattice vectors share an analogous relationship with those of the reciprocal superlattice. The reciprocal lattice vectors of a direct lattice are calculated by

$$\begin{bmatrix} \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3 \end{bmatrix}^T = [\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3]^T \quad (4)$$
where the vectors share the same definition as in equations (1) and (2). Similarly, the primitive reciprocal lattice vectors of the superlattice can be obtained by

\[
[d_1, d_2, d_3]^T = [c_1, c_2, c_3]^{-1}
\]

(5)

where \(d_1\), \(d_2\), and \(d_3\) are the reciprocal lattice vectors corresponding to the direct superlattice. Substituting equations (4) and (5) into equation (2), the following relationship can be derived:

\[
[b_1, b_2, b_3]^T = M^T [d_1, d_2, d_3]^T.
\]

(6)

The matrix multiplication order implies that the row vectors of the matrix \(M^T\) contain the coordinates of the vectors \(\{b_1, b_2, b_3\}\) in the basis of \(\{d_1, d_2, d_3\}\).

In terms of the matrix \(M\), equation (1) can be written as

\[
k = (n_1, n_2, n_3) \begin{pmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{pmatrix}^{-1} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}^T + s
\]

\[
= (n_1, n_2, n_3)(M^{-1})^T (b_1, b_2, b_3)^T + s
\]

\[
= (n_1, n_2, n_3)[d_1, d_2, d_3]^T + s
\]

(7)

Therefore, the set of vectors \(\{d_1, d_2, d_3\}\) are a generating basis of the \(k\)-point grid. As shown in equation (7), the traditional Monkhorst-Pack scheme uses a diagonal matrix \(M\), which is equivalent to the constraint that the \(k\)-point grids are aligned with the reciprocal lattice vectors. However, Froyen has pointed out that this constraint is not necessary [5], and we have previously demonstrated that much more efficient grids can be generated if the Monkhorst-Pack approach is generalized by relaxing this requirement [4]. Whether or not \(M\) is diagonal, the volume ratio between a unit cell in the superlattice and a primitive cell is equal to the absolute value of the determinant of \(M\). We will refer to the determinant as the “size” of the corresponding superlattice. The size also equals the number of total \(k\)-points in the first Brillouin zone of the corresponding \(k\)-point grid.

The isomorphism between a \(k\)-point grid and a direct superlattice suggests that the minimum spacing between points on a superlattice, \(r_{\text{lat}}\), is a good descriptor of the accuracy of a density functional theory (DFT) calculation using the corresponding \(k\)-point grid [5]. A rigorous justification for the adoption of
this descriptor for semiconductors and insulators has been laid out in our previous work [4], where we showed that the integration error due to approximating the integral by a discrete sum over \( k \)-points is expected to decay exponentially with a decay rate approximately proportional to \( r_{\text{lattice}} \). This insight is supported by a recent empirical study by Choudhary and Tavazza which indicates that this length-based descriptor for \( k \)-point density has smoother convergence in DFT calculations and stronger correlations with physical parameters of materials over a measurement based on the total number of \( k \)-points per reciprocal atom [6]. Based on this insight, we developed a publicly accessible server that returns to the user the optimal generalized \( k \)-point grid that minimizes the number of irreducible \( k \)-points (computational cost) for a given value of \( r_{\text{lattice}} \) (desired accuracy) [4]. The server is backed by a database of pre-calculated grids for all systems other than triclinic and monoclinic, for which generalized \( k \)-point grids are generated dynamically. The use of this approach has been shown to reduce the computational cost for well-converged calculations by a factor of approximately 2 compared to regular Monkhorst-Pack grids [4, 7].

There has been increasing interest in the generation and use of generalized \( k \)-point grids [6, 8-23]. Of particular note, Hart and co-workers have recently developed algorithms and released an open-source software package (GRkgridgen) for generating generalized \( k \)-point grids on the fly [24, 25]. Some of the core elements of these algorithms, such as the use of Hermite Normal Form to identify unique superlattices [26, 27], are in similar to some of ours. However, there are some notable differences. In particular, we have developed algorithms that perform an exhaustive search for the symmetry-preserving grid that minimizes the number of irreducible \( k \)-points (computational cost) for a user-provided lower bound for \( r_{\text{lattice}} \), which has been shown both theoretically [4] and empirically [6] to be a good descriptor of grid accuracy. We present algorithms for both on-the-fly grid generation and a database lookup that are optimized to accomplish this efficiently. In addition, we present an algorithm for identifying low-dimensional structures, so that the \( k \)-point grids can be reduced accordingly.

To facilitate the use of generalized \( k \)-point grids, we are releasing two open-source tools to the community. For the first tool, we have packaged our server into an open-source stand-alone application that is available at [https://gitlab.com/muellergroup/k-pointGridGenerator](https://gitlab.com/muellergroup/k-pointGridGenerator). This application is self-contained and written in Java, and it can be run without the need for compilation or external library installation on any machine on which Java is installed. Like the server, the stand-alone application dynamically generates grids for monoclinic and triclinic crystal systems and uses a database of pre-calculated grids for all others. This application can be downloaded and run locally without the need for an internet connection at run time. It contains all of the features of the server, including automatic slab detection. We have also developed
kpLib, a lightweight library written in C++ that dynamically generates grids on the fly, requiring only that the users provide information about the space group of the reciprocal lattice. This library is designed for integration with third-party applications and to be compatible with external software packages such as Spglib [29]. To maximize the efficiency of the tools, the algorithms applied are optimized based on a user-provided lower bound for $r_{\text{lattice}}$. In the following sections, a detailed explanation of the core algorithms underlying these tools is provided.

2. Methodology

In this section we discuss the core algorithms behind our tools. We start by describing our approach for dynamically generating generalized grids, as this approach is used to populate the database, and is adopted for monoclinic and triclinic crystal systems in all implementations, and for all grid generation by kpLib. Then we describe the algorithms for enumerating symmetry-preserving superlattices and for determining the symmetrically irreducible $k$-points and corresponding weights. We also provide overviews of how grids are stored in and retrieved from the database, and how we automatically detect systems with fewer than three periodic dimensions and adjust the grids accordingly. For conciseness we use the symbol $N_i$ and $N_f$ to represent, respectively, the number of symmetrically irreducible $k$-points and the number of total $k$-points in the Brillouin zone. $N_f$ is also then the number of primitive cells in a unit cell of the corresponding real-space superlattice (aka the “size” of the superlattice).

2.1 An Overview of the Dynamic Grid Generation Algorithm

The grid generation algorithm starts with three parameters describing the input structure:

1. The real-space primitive lattice vectors, $\{a_1, a_2, a_3\}$.

2. The real-space conventional lattice vectors, where at least one of the vectors is orthogonal to the other two for all but triclinic systems. We label these $\{c_1, c_2, c_3\}$.

3. The group of point symmetry operations, $\{R\}$, that the $k$-point grid (and real-space superlattice) should preserve. These point symmetry operations can be generated by removing translation from
all the operations in the real-space crystallographic space group. If the system has time reversal symmetry, then the reciprocal-space band structure will have inversion symmetry even if the real-space crystal does not. In this case, inversion and any additional operators required to complete the group should be added if they are not already present.

The algorithm searches for the \( k \)-point grid that minimizes \( N_i \) while satisfying the following two constraints:

1. \( r_{\text{lattice}} \) for the corresponding superlattice not smaller than \( r_{\text{min}} \) (a value provided by the user),

2. \( N_r \) is greater than or equal to \( N_{\text{min}} \) (another value provided by the user).

Symmetry-preserving lattices, in which every symmetry operation maps a lattice point to another lattice point, usually result in the smallest values of \( N_i \) by maximizing the use of symmetry. Thus, the algorithm only searches over lattices that are symmetry-preserving.

We start by determining a lower bound for \( N_r \), which we call, \( N_{\text{lower}} \). It is the larger value of \( N_{\text{min}} \) and the minimum size that any superlattice can have while satisfying \( r_{\text{lattice}} \geq r_{\text{min}} \):

\[
N_{\text{lower}} = \max \left( N_{\text{min}}, \left\lfloor \frac{\sqrt{2}}{2} r_{\text{min}}^3 / V_p \right\rfloor \right)
\]

where \( V_p \) is the volume of the primitive cell, \( \frac{\sqrt{2}}{2} r_{\text{min}}^3 \) is the volume of unit cell in a face-centered cubic (fcc) structure, \( \lfloor x \rfloor \) is the floor operation that returns the largest integer no greater than the argument \( x \), and \( \max() \) is a function that returns the argument with the maximum value. Equation (8) can be justified by considering that fcc structures maximize the packing density for rigid spheres [30] and thus \( \frac{\sqrt{2}}{2} r_{\text{min}}^3 \) is the minimum unit cell volume for a superlattice of which \( r_{\text{lattice}} \) is at least \( r_{\text{min}} \).

The search for optimal superlattices starts with lattices of size \( N_{\text{lower}} \) and generates symmetry-preserving superlattices using an algorithm to be introduced in section 2.3. For each symmetry-preserving superlattice, the scheme checks whether \( r_{\text{lattice}} \) is smaller than \( r_{\text{min}} \) and discards it if it is. When the first
superlattice for which $r_{\text{lattice}} \geq r_{\text{min}}$ is found, its corresponding $k$-point grid is kept as the initial “best grid”, and the scheme can determine an upper limit for the search, $N_{\text{upper}}$:

$$N_{\text{upper}} = N_i \times N_{\text{sym}}$$  \hspace{1cm} (9)

where $N_{\text{sym}}$ is the number of unique point symmetry operations for the system, as provided in the third input parameter listed above. Any superlattices with $N_T \geq N_{\text{upper}}$ would necessarily have more irreducible $k$-points than that of the initial best grid. If at some point a superlattice with $N_i$ smaller than that of the best known grid is found, the best grid is updated to this newly found one and the value of $N_{\text{upper}}$ is adjusted accordingly. When two $k$-point grids have the same $N_i$, the scheme favours the one with a larger $r_{\text{lattice}}$ in the corresponding superlattice. If $r_{\text{lattice}}$ of both superlattices also tie, the scheme chooses the one with a larger $N_T$. The search ends when the upper limit of the sizes of superlattices is reached. Figure 1 summarizes the steps of the scheme.

Figure 1. A diagram summarizes the workflow of the dynamic grid generation algorithm.
2.2 Representation of Superlattices and Identification of Symmetry-Preserving Superlattices

Each superlattice is represented by the Hermite normal form (HNF) [26, 27] of the transformation matrix, \( \mathbf{M} \), where \( \mathbf{M} \) is as shown in equation (3). The Hermite normal form (\( \mathbf{H} \)) is defined for integral matrices and satisfies the following requirements

\[
H_{ij} = 0, \text{ if } j > i
\]

\[
0 \leq H_{ij} < H_{ij}, \text{ if } j < i
\]

Equation (10) states that a matrix in HNF is lower-triangular, and equation (11) requires that all elements are non-negative and the maximum element in each column resides on the diagonal. There is an equivalent upper-triangular formulation of the HNF of a matrix, but we use the lower-triangular one. Each non-singular integral matrix can be transformed into its Hermite normal form by multiplying a series of unimodular matrices (integral matrices whose determinant is 1 or -1). In other words, the determinant of a matrix and its HNF are equal. And it has been shown that two superlattices are equivalent if and only if the HNF of their generation matrix \( \mathbf{M} \) are the same [26, 27]. An alternative, yet equivalent statement is that the HNF of a matrix is unique. The uniqueness of HNF of a matrix provides a convenient algorithm to enumerate all possible superlattices of a primitive lattice. The details have been laid out in references [26, 27]. We use a simple algorithm, developed from the uniqueness of the HNF of a matrix, to determine whether a superlattice preserves a given point symmetry operation. The algorithm works as follows:

1) Multiply the matrix representation of the point symmetry, \( \mathbf{R} \), with the HNF, \( \mathbf{H} \), of the transformation matrix, \( \mathbf{M} \).

2) Find the HNF form of the resulting integral matrix from last step, \( \mathbf{H}' \).

3) If \( \mathbf{H} = \mathbf{H}' \), then this superlattice preserves this point operation \( \mathbf{R} \). Otherwise, the superlattice doesn’t possess this symmetry.

2.3 Algorithm for Efficient Enumeration of Symmetry-Preserving Superlattices

The optimality of the returned GMP grid relies on the exhaustiveness of the enumeration of all possible superlattices. Such computationally expensive enumeration has been identified as the main hurdle of applying GMP grids in calculations of properties of crystalline materials [4, 31]. A rough estimation can show how the number of possible superlattices grows with lattice size. Given a size, \( N_T \), of a superlattice,
and a factorization of it into three distinct positive integers \(\{N_1, N_2, N_3\}\), the number of unique matrices in HNF with the three numbers as the diagonal elements, would be

\[
N_1^2 N_2 + N_2^2 N_1 + N_1^2 N_3 + N_3^2 N_1 + N_2^2 N_3 + N_3^2 N_2,
\]

counting all permutations of the three numbers at the diagonal positions. The total number of possible superlattices can be calculated by considering all ways in which \(N_r\) can be factored into three numbers. For example, the total number of unique superlattices with a size of 1000, which is a typical number of the total number of \(k\)-points in calculations of metallic systems, is 3,147,430.

Morgan et al. have presented an algorithm for accelerating the enumeration of symmetry-preserving lattices for a given lattice size by expressing the primitive lattice in Niggli-reduced form [25]. For each of the 44 distinct Niggli bases, they have determined symmetry-based constraints on the entries of \(H\) that can be used to reduce the number of possible lattices that must be considered. We have developed an approach that similarly iterates over symmetry-preserving lattices, with two key differences: it does not rely on Niggli reduction, and it is optimized for grid selection based on \(r_{min}\), which has been shown to work well as a descriptor of \(k\)-point density both in theory [4] and in practice [4, 6].

### 2.3.1 Crystal Systems Other than Triclinic

We start by considering systems that are not triclinic. For such systems at least one of the conventional lattice vectors must, by the symmetry of the system, be perpendicular to the other two. For simplicity, our only requirement is that such a vector be listed third, as \(c_3\).

The key to our approach is the recognition that any regular three-dimensional lattice consists of layers of identical two-dimensional lattices that are normal to \(c_3\). Each two-dimensional lattice may be shifted from the one below it by a constant shift vector that is parallel to its lattice plane, and for symmetry-preserving lattices only a finite set of shift vectors are allowed. For example, if there is a twofold rotational axis parallel to \(c_3\), then this axis may only pass through a finite set of points in the two-dimensional lattice. Any other shift would result in a lattice that is not symmetry preserving. Similarly, if there is a mirror plane perpendicular to \(c_3\), then either the mirror plane must base at the mid-point between two layers, in which case no shift is allowed, or it must pass through one of the layers, in which only the shifts shown in Table 1(all half-shifts of lattice vectors) are allowed. This concept is illustrated in two dimensions in Figure 2. Similar sets of shifts may be derived for three-fold rotational axes (Table 1).
Table 1. Possible displacements of lattice planes in real space in 2 dimensions, and of the Γ point in reciprocal space in 3 dimensions.

| Crystal System     | Shift vectors in the basis of \{c_1, c_2\} in real space | Shift vectors of the Γ point in the basis of \{d_1, d_2, d_3\} as defined in equation (5) |
|--------------------|-----------------------------------------------------------|-----------------------------------------------------------------------------------------|
| Cubic, Tetragonal, Orthorhombic, Monoclinic | [0.0, 0.0], [0.0, 0.5], [0.5, 0.0], [0.5, 0.5] | [0.0,0.0,0.0], [0.0,0.0,0.5], [0.0,0.5,0.0], [0.5,0.0,0.0], [0.5,0.5,0.0], [0.5,0.0,0.5], [0.0,0.5,0.5], [0.5,0.5,0.5] |
| Hexagonal, Trigonal | [0.0, 0.0], [1/3, 0.0], [0.0, 1/3], [0.0, 2/3], [2/3, 0.0], [1/3, 1/3], [2/3, 2/3], [1/3, 2/3], [2/3, 1/3] | |

Figure 2. Two-dimensional examples of allowed and disallowed shifts. In all examples the blue line represents a mirror plane, black dots represent lattice points on the real-space superlattice, and the dashed lines show the different layers of lattice points that are orthogonal to \(c_2\) (which here serves the role that \(c_3\) serves in three dimensions). a), b), c), and d) show allowed shifts in which the mirror plane transforms every lattice point to another lattice point. In a) and b) there is zero shift, and in c) and d) the shift is half the real-space superlattice vector normal to \(c_2\). e) and f) show disallowed shifts, as the mirror planes do not transform every lattice point to another lattice point.
A high-level summary of our algorithm for enumerating symmetry-preserving lattices is then as follows:

1) Determine all pairs of factors of the total lattice size. In each pair, the first factor represents the size of the supercell in each two-dimensional layer and the second represents the number of layers in each three-dimensional supercell.

2) For each pair of factors, enumerate all symmetry-preserving two dimensional lattices with the required size.

3) Combine each two-dimensional lattice with each allowed shift to create a candidate three-dimensional lattice.

4) Verify that the three-dimensional lattice is symmetry-preserving as described above.

This algorithm effectively reduces the problem of enumerating three-dimensional lattices to one of enumerating two-dimensional lattices, which significantly accelerates the search for symmetry-preserving lattices. We can further accelerate the search by recognizing that if the number of layers is too small to satisfy the requirement that \( r_{lattice} \geq r_{min} \), we can skip the enumeration of two-dimensional lattices and move on to the next set of factors. Similarly, if we ever determine that the \( r_{lattice} < r_{min} \) for any two-dimensional layer, then we can stop evaluation of all lattices constructed from that layer and move onto the next two-dimensional lattice. We find that pre-screening the lattices for \( r_{lattice} \) in this way significantly increases the speed of the algorithm when \( r_{min} \) is the limiting factor, as demonstrated in benchmarking calculations in section 4.1.

The steps of the algorithm are shown by the pseudocode in Figure 3. Some elaborations of the pseudocode are given below:

- The input conventional vectors should have \( \mathbf{e}_3 \) orthogonal to the other two vectors. In addition, if the unit cell defined \( \mathbf{e}_1 \) and \( \mathbf{e}_2 \) contains a centring point, we replace them with a pair of primitive lattice vectors. In other word, \( \{\mathbf{e}_1,\mathbf{e}_2\} \) always define a primitive lattice in its residing lattice plane to ensure the algorithm can exhaustive all possible superlattices by multiplying integral matrices.
Algorithm 1 Fast Enumeration of Symmetry-Preserving Superlattices

Input:
- $N_T$ - Size of superlattice. Also the total number $k$-point in the grid.
- $\{c_1, c_2, c_3\}$ - Conventional lattice vectors.
- $\{R\}$ - The point symmetry group to be preserved in superlattices.
- $\{a_1, a_2, a_3\}$ - Primitive lattice vectors.
- $s$ - Array of possible 2D shift vectors.
- $r_{min}$ - Minimum periodic distance specified by users.

Output:
- $\{\{g_1, g_2, g_3\}\}$ - Array of symmetry-preserving superlattices with $r_{lattice} \geq r_{min}$.

Initialization:
- $\{\{g_1, g_2, g_3\}\}$ - Empty array of symmetry-preserving superlattices.
- M - A $3 \times 3$ zero matrix.

1: find the point symmetry group, $\{R^{2D}\}$, of the 2D sublattice $\{c_1, c_2\}$;
2: factorSets[] ← sets of factorizations of $N_T$ into 2 integer numbers;
3: for factors[] in factorSets[] do
4:     primLayerSpacing = $\|c_3\| \times V_y/V_x$;
5:     maxLayers = isHexagonal() ? 3 : 2;
6:     maxZDistance = factors[2] × primLayerSpacing × maxLayers;
7:     if maxZDistance < $r_{min}$ then
8:         continue;
9:     \{\{g_1, g_2\}\} ← initialize an empty array of 2 vectors;
10:    factorSets2D[] ← sets of factorization of factors[1] into 2 integral numbers;
11: for factors2D[] in factorSets2D[] do
12:     $M_{11} = $ factors2D[1], $M_{22} = $ factors2D[2]
13: for i = 0 to factors2D[1] - 1 do
14:     $M_{21} = i$;
15:     if not symmetryPreserving($M, \{R^{2D}\}$) then
16:         continue;
17:     $g_1 = M_{11} c_1$;
18:     $g_2 = M_{12} c_1 + M_{22} c_2$;
19:     add $\{g_1, g_2\}$ to the array of 2D symmetry-preserving superlattices, $\{\{g_1, g_2\}\}$;
20: for $\{g_1, g_2\}$ in $\{\{g_1, g_2\}\}$ do
21:     if getMinDistance($g_1, g_2$) < $r_{min}$ then
22:         continue;
23: for s in $\{s\}$ do
24:     $g_3 = [s_1, s_2] \cdot [g_1, g_2]^T + (\text{factors}[2]/(V_y/V_x)) \cdot c_3$;
25:     if getMinDistance($g_1, g_2, g_3$) < $r_{min}$ then
26:         continue;
27:     $M = [g_1, g_2, g_3]^T \cdot ([a_1, a_2, a_3]^T)^{-1}$;
28:     if $M$ contains non-integral elements then
29:         continue;
30:     if not symmetryPreserving($M, \{R\}$) then
31:         continue;
32:     add $\{g_1, g_2, g_3\}$ to the array of symmetry-preserving superlattices;
33: return $\{\{g_1, g_2, g_3\}\}$.

Figure 3. Algorithm for fast enumeration of symmetry preserving superlattices for systems other than triclinic.
• Line 2: Permutations of the same set of 2 factors are treated as different factorizations, i.e. \{2, 3\} and \{3, 2\} are two different factorizations of 6.

• The for loops in line 3, line 11, line 20 and line 23 means for each of the elements in the set, execute the loop once. At the end of each execution of the loop, the looping element iterate to the next one. The “continue” in line 8, 16, 22, 26, 29 and 31 means skipping the remaining operations in the current loop and continuing the loop with the next element in the set.

• Line 4 – line 6: this block calculates the maximum possible length of the shortest vector parallel with \(\mathbf{c}_i\) in a superlattice. \(V_c\) and \(V_p\) are volumes of the conventional and primitive unit cells, respectively. “primLayerSpacing” is the distance between adjacent layers of two-dimensional lattices. “maxLayers” is the maximum number of layers after which the stacking sequence starts to repeat. This value is 3 for lattices in hexagonal crystal family (including hexagonal and trigonal lattices) and 2 for other lattices. The reason for the differences is that lattices have different sets of two-dimensional shifts, as listed in Table 1.

• “isHexagonal() ? 3 : 2” represents a conditional operator and is an abbreviation of the statement “if isHexagonal() == true then 3 else 2”.

• Line 6: arrays in the pseudocode have their indices starting from 1. Therefore, “factor[2]” indicates the second factor of this factorization.

• Line 15 and line 30: the algorithm discussed in section 2.2 is used to determine whether a superlattice preserves the given group of point symmetries.

• Line 21 and line 25: “getMinDistance(\(\{g_i\}\))” is a function that uses Minkowski reduction [32] to determine the minimum distance between two lattice points in the lattice defined by \(\{g_i\}\). The input lattice can have any dimensionality up to three.

• Line 28: if \(\mathbf{M}\) has non-integral elements, then \(\{g_1, g_2, g_3\}\) is not a superlattice of the primitive lattice.

In addition to pre-screening for \(r_{\text{lattice}}\), this algorithm accelerates the enumeration in several ways. Firstly, it drastically decreases the total number of 3-dimensional superlattices that need to be checked for symmetry preservation. Secondly, the groups of symmetry in 2-dimensional sublattice have fewer number
of symmetry operations comparing with their corresponding groups in 3 dimensions. And thirdly, a 2-
dimensional matrix multiplication takes fewer elementary operations than a 3-dimensional one.

Algorithm 2 Fast Enumeration of Symmetry-Preserving Superlattices for Triclinic Structures

**Input:**
- \( N_T \) - Size of the superlattice. Also the total number \( k \)-point in the grid.
- \( r_{min} \) - Minimum periodic distance specified by users.
- \( \{ a_1, a_2, a_3 \} \) - Primitive lattice vectors.

**Output:**
- \( \{ \{ g_1, g_2, g_3 \} \} \) - Array of symmetry-preserving superlattices with \( r_{lattice} \geq r_{min} \).

**Initialization:**
- \( H \) - A 3 \( \times \) 3 zero matrix.
- \( \{ \{ g_1, g_2, g_3 \} \} \) - Empty array of symmetry-preserving superlattices.

1: factorSets[] \( \leftarrow \) sets of integral factorizations of \( N_T \) into 3 numbers;
2: for \( \{ N_1, N_2, N_3 \} \) in factorSets[] do
3: \( H \leftarrow \) put \( N_1, N_2, N_3 \) on diagonal positions;
4: \( g_1 = H_{11} \cdot a_1 \);
5: if \( ||g_1|| < r_{min} \) then
6: continue;
7: for \( H_{21} = 0 \) to \( N_1 - 1 \) do
8: \( g_2 = H_{31} \cdot a_1 + H_{32} \cdot a_2 \);
9: if getMinDistance(\( g_1, g_2 \)) < \( r_{min} \) then
10: continue;
11: for \( (H_{31}, H_{32}) = (0, 0) \) to \( (N_1 - 1, N_2 - 1) \) do
12: \( g_3 = H_{31} \cdot a_1 + H_{32} \cdot a_2 + H_{33} \cdot a_3 \);
13: if getMinDistance(\( g_1, g_2, g_3 \)) < \( r_{min} \) then
14: continue;
15: add \( \{ g_1, g_2, g_3 \} \) to the array of superlattices;
16: return the array of superlattices;

Figure 4. Algorithm for enumerating symmetry-preserving superlattices for triclinic system, accelerated by enforcing \( r_{lattice} \geq r_{min} \) at each dimension.

2.3.2 Triclinic Crystal System

The triclinic system doesn’t benefit from the above algorithm since all its superlattices preserve the point symmetry operations of the primitive lattice, namely the identity operation and sometimes the inversion operation. For triclinic systems we accelerate the search for superlattices for which \( r_{lattice} \geq r_{min} \) by again considering one dimension at a time. For each factor set, if \( H_{11} |a_1| < r_{min} \), the shortest distance between lattice points must be less than \( r_{min} \) and the factor set is not considered. Similarly, if the two dimensional lattice spanned by \( H_{11}a_1 \) and \( H_{21}a_1 + H_{22}a_2 \) has \( r_{lattice} < r_{min} \) we do not iterate over possible values of \( H_{31} \) and \( H_{32} \) as we already know the lattices will not satisfy the required constraint. The procedures are summarized as a pseudo code in Figure 4. The input lattice can be of any dimension up to three. We note that a similar approach can be used for any scheme based on iterating over lattices in HNF, such as the one developed by Morgan et al. [25].
2.4 Evaluating shift vectors

$K$-point grids can be generated for each symmetry-preserving lattice using equation (7), where the matrix $H$ can be used for $M$. The only remaining unknown in equation (7) is the shift vector $s$. For a shift to be guaranteed to result in a symmetry-preserving lattice, it must shift the origin to a point that has the full point group symmetry of the origin. For all symmorphic space groups, the only such points are located at linear combination of full- or half-multiples of the primitive lattice vectors. Thus, we consider only the eight such unique combination of $k$-point grid generating vectors, $\{d_1, d_2, d_3\}$, as candidate shift vectors (Table 1). In some cases (e.g. hexagonal systems), some of the shift vectors in Table 1 will not result in a symmetry-preserving grid. We identify and reject these when determining the number of irreducible $k$-points. As this occurs as soon as the first point that breaks symmetry is encountered, it comes with relatively little computational cost.

2.5 Algorithms for fast calculation of symmetrically irreducible $k$-points and $k$-point weights

We select the optimal lattice based on the values of $N_i$, $r_{\text{lattice}}$, and $N_T$. The value of $r_{\text{lattice}}$ can be easily obtained from the superlattice vectors by Minkowski reduction, and $N_T$ equals the absolute value of the determinant of the transformation matrix $M$. However, calculating $N_i$ for a $k$-point grid is a relatively expensive operation. An intuitive approach is to apply all the point symmetry operations to each $k$-point, $k_j$, and compare the resulting coordinates with all the other $k$-points. If one of the transformed $k$-points, $k_j'$, is translationally equivalent to one of the other $k$-points, $k_j$, then the $k$-points $k_i$ and $k_j$ are symmetrically equivalent. However, this algorithm scales by $O(N_T^2)$, where $N_T$ is the number of total $k$-points of a grid. As this operation is applied to each of the $k$-point grids found by the algorithm in section 2.3, this intuitive but costly approach could easily become the major overhead of any $k$-point generation scheme.

We solve this complication by first recognizing that a unit cell in reciprocal space is a supercell of a regular $k$-point lattice, where the two lattices are related by equation (6). The matrix $M^T$ is the transpose of the matrix used to construct the real-space superlattice that corresponds to the $k$-point grid. We transform $M^T$ to Hermite normal form. To avoid confusion with the Hermite normal form of $M$, which we have labelled...
H, we will refer to the Hermite normal form of $M^T$ as $J$. We note that in general, $J$ does not have to be equal to $H^T$. The key to our approach is the recognition that it is possible to tessellate all of reciprocal space with unit cells of size $J_{11} \times J_{22} \times J_{33}$ arranged periodically on the superlattice, where $J_{11}$, $J_{22}$, and $J_{33}$ are the diagonal elements of $J$. This is illustrated in two dimensions in Figure 5, but the same concept extends to any number of dimensions. The off-diagonal elements of $J$ serve to shift each layer of unit cells relative to the previous layer, so that the tessellation resembles stacked bricks. Within each of these unit cells, the coordinates of a $k$-point can be expressed as:

$$\mathbf{r} + ([k_1, k_2, k_3] + s) \cdot [d_1, d_2, d_3]^T$$

where $\mathbf{r}$ is a lattice point on the reciprocal space lattice (blue dots in Figure 5), $d_1$, $d_2$, and $d_3$ are generating lattice vectors of the $k$-point lattice (also reciprocal primitive lattice vectors), $k_1$ is an integer from 0 to $J_{11} - 1$, $k_2$ is an integer from 0 to $J_{22} - 1$, and $k_3$ is an integer from 0 to $J_{33} - 1$. The coordinates of the $k$-point can then be easily transformed into any basis (such as that of the primitive lattice in reciprocal space) using linear operations. We have shared this approach for iterating over $k$-points with the Hart group for their work with generalized $k$-point grids [24]. Values for $k_1$, $k_2$, and $k_3$ can be quickly calculated for any $k$-point using integer arithmetic, as shown in Figure 6, line 15 and discussed below.

Given the enumeration of $k$-points using equation (13), we identify irreducible $k$-points in a way similar to that described by Hart et al. [24]. We assign a unique index to each $k$-point in the Brillouin zone or, equivalently, to each $k$-point in any unit cell of the reciprocal lattice. The values of the index range from 1 to $N_T$, and translationally equivalent $k$-points share the same index. Linear scaling is achieved because the index for any given $k$-point can be calculated in constant time, as can the sublattice of $k$-points that have a given index. Indices are assigned to $k$-points using

$$index = 1 + k_1 + J_{11} k_2 + J_{11} J_{22} k_3.$$  

(14)

Then iteration of all $k$-points in a unit cell in reciprocal space, equivalent to all $k$-points in the Brillouin zone, is accomplished by looping over values of $k_1$, $k_2$, and $k_3$ in equation (14).
Figure 5. Two-dimensional illustrations of the concepts used for k-point enumeration and index generation. The top row provides the three possible matrices in Hermite normal form for the set of factors (3,2). The middle row shows the three Bravais superlattices corresponding to these matrices, assuming that the generating vectors for the k-point grid, $\mathbf{d}_1$ and $\mathbf{d}_2$, are aligned with the dashed gray lines. The bottom row shows how space can be tessellated by unit cells that are $3 \times 2$ supercells of the generating lattice vectors, where each unit cell is associated with a point on the superlattice (in the lower-left corner of the cell). The k-point indices within each cell, as would be calculated by the two-dimensional equivalent of equation (14), are shown.

To count the number of distinct k-points, we iterate over all translationally distinct k-points as described above and apply all symmetry operations to each k-point. If an operation does not transform the k-point to another k-point, the grid is not symmetry-preserving and is rejected (this can sometimes happen if a shift breaks symmetry). If the index of any symmetrically equivalent k-point is less than that of the current k-point, then we have already seen a symmetrically equivalent k-point, so the counter for the number of irreducible k-points is not incremented. If there is no symmetrically equivalent k-point with an index lower than that of the current k-point, then the current k-point is the first we’ve seen in its orbit, so the counter for the number of irreducible k-points is incremented. A simple variation of this algorithm is used to calculate k-point weights by, for each k-point, determining the orbit of symmetrically equivalent points and then incrementing the weight of the k-point that has the lowest index in that orbit. Figure 6 provides the pseudocode of this algorithm.
Algorithm 3: Fast calculation of symmetrically irreducible \( k \)-points and \( k \)-points weights

Input:
- \( \{ R \} \) - The point symmetry group of the symmorphic space group of input structure.
- \( H \) - The transformation matrix of real-space superlattice in Hermite normal form.
- \( N_T \) - Number of total \( k \)-points for the grid to be calculated for.
- \( s \) - Shift vector of the \( \Gamma \) point.

Output:
- \( N_s \) - Number of symmetrically irreducible \( k \)-point.
- coordinates\[N_T\] - Array of \( k \)-point coordinates.
- weights\[N_T\] - Array of weights.

Initialization:
- \( \{ R' \} \) - Empty array of 3x3 matrices, representing symmetry operation in the basis of generating vectors, \( \{ d_1, d_2, d_3 \} \).
- \( N_s = 0 \).
- coordinates\[N_T\] - Zero array of coordinates.
- weights\[N_T\] - Zero array.

1: for \( R \) in \( \{ R \} \) do
2: \( R' = H^T \cdot (R^T)^{-1} \cdot (H^T)^{-1}; \)
3: add \( R' \) to the array, \( \{ R' \} \);
4: \( J \leftarrow \text{HNF of } H^T; \)
5: let \( J = [j_1, j_2, j_3]^T; \)
6: for \( k_3 = 0 \) to \( J_{33} - 1 \) do
7: \( k_2 = 0 \) to \( J_{22} - 1 \) do
8: \( k_1 = 0 \) to \( J_{11} - 1 \) do
9: \( \text{index} = 1 + k_1 + J_{11}k_2 + J_{11}J_{22}k_3; \)
10: \( \text{minIndex} \leftarrow \text{initialize as } \text{infinity}; \)
11: for \( R' \) in \( \{ R' \} \) do
12: \( [k'_1, k'_2, k'_3] = [(k_1, k_2, k_3) + s] \cdot R' - s; \)
13: if \( [k'_1, k'_2, k'_3] \) contains non-integral values then
14: return \( 0, \text{null, null}; \)
15: \( [k''_1, k''_2, k''_3] = [k'_1, k'_2, k'_3] - \sum_{i=1}^{3} ([k'_i/J_{ii}] \cdot j_i); \)
16: \( \text{newIndex} = 1 + k''_1 + J_{11}k''_2 + J_{11}J_{22}k''_3; \)
17: \( \text{minIndex} \leftarrow \text{min}(\text{minIndex, newIndex}); \)
18: \( [x_1, x_2, x_3] = ([k_1, k_2, k_3] + s) \cdot (H^T)^{-1}; \)
19: \( k = [x_1 \% 1, x_2 \% 1, x_3 \% 1]; \)
20: coordinates\[index\] = \( k; \)
21: if \( \text{minIndex} == \text{index} \) then
22: \( N_s = N_s + 1; \)
23: weights\[index\] = 1;
24: else
25: weights\[minIndex\] = weights\[minIndex\] + 1;
26: return \( N_s, \) coordinates\[], weights\[];

Figure 6. Algorithm for fast calculation the coordinates of symmetrically irreducible \( k \)-points and the corresponding weights.

Some explanations are provided as follows:

- Line 12 – line 14: this code block determines whether a given shift vector preserves all the point symmetries. The idea is that the resulting \( k \)-point, from applying firstly a shift and then a symmetry operation, should also be a \( k \)-point that is generated by shifting another \( k \)-point. Hence if the
coordinates of the resulting point expressed in the basis of the \( k \)-point lattice, \([k'_1, k'_2, k'_3]\), are not integers then the point is not a point on the grid. As soon as the first point that does not preserve symmetry is identified, the algorithm returns empty results and terminates the remaining steps.

- Line 15: this is the reverse operation of equation (13) in the basis of the reciprocal primitive lattice. Given a \( k \)-point, it finds the integral coordinates of the translationally equivalent \( k \)-point within the unit cell located at the origin. \( \left\lfloor k'_i / J_{ii} \right\rfloor \) represents the floor operation, which returns the greatest integer no larger than the argument.
- Line 18 – line 20: changes the basis of the \( k \)-point coordinates to the reciprocal superlattice vectors, \( \{b_1, b_2, b_3\} \), which is the reciprocal lattice of the input primitive lattice.
- Line 19: “\( \% \)” represents the modulo operation. This operation shifts all the coordinates to their equivalent ones within the supercell defined by \( \{b_1, b_2, b_3\} \).
- Line 26: the returned arrays contain coordinates and weights for all \( k \)-point. The symmetrically non-distinct points, however, have weights of zero. This fact is used to identify the distinct ones from returned array.

### 2.6 Algorithm for detection of structures without 3-dimensional periodicity

In software packages that assume three-dimensional periodicity for all calculations, low-dimensional structures such as surfaces and nanoparticles are modelled by adding vacuum to the normal directions of the periodic low-dimensional lattice. We will refer to such normal directions as the “vacuum directions”. As there is little interaction between materials separated by vacuum, it is not necessary to sample more than one \( k \)-point in reciprocal lattice directions that are normal to the real-space periodic lattice. To ensure efficient \( k \)-point grids are generated in such cases, we have developed an algorithm to determine when structures are suitably separated by vacuum, and we adjust the generated \( k \)-point grid accordingly. For example, when simulating a slab, the density of the grid will be minimized along the direction parallel to the vacuum direction. For nanoparticles, only a single \( k \)-point will be returned.
Figure 7. Two examples of low-dimensional systems recognized by our algorithm. A) A slab with adsorbed molecules. As long as the distances between all atom in one slab (including adsorbates) and the nearest atoms in a neighboring slab (including adsorbates) is at least $r_{\text{gap}}$, this will be treated as a low-dimensional system. B) An example with one-dimensional chains oriented in different directions. As long as the distances between chains are at least $r_{\text{gap}}$, the algorithm will recognize this system as being periodic in two dimensions.

The user-provided input to our algorithm is a minimum distance by which slabs, nanowires, or surfaces must be separated to trigger a change in the $k$-point grid. We call this quantity $r_{\text{gap}}$. Given a value for $r_{\text{gap}}$, our algorithm identifies gaps between slabs, nanowires, or nanoparticles regardless of the topology of the system (Figure 7). We accomplish this by starting at a single atom and recursively visiting all neighbouring atoms within a distance of $r_{\text{gap}}$. When we encounter an atom that is translationally equivalent to an atom we have already visited, we record the vector between those atoms. We refer to such vectors, which are normal to the vacuum directions, as “contiguous vectors”. For a slab structure, the contiguous vectors will be parallel to the slab surface. In a nanowire, the contiguous vector is the lattice vector parallel with the nanowire. The contiguous vectors are not necessarily the input primitive lattice vectors but must be linear combinations of them. In some cases, (e.g. a molecule between two slabs, or something like Figure 7b), there may be more than one set of contiguous atoms that are separated by at least $r_{\text{gap}}$. We identify such cases by ensuring that we have visited each set of translationally equivalent atoms at least once.

This algorithm is shown by the pseudocode in Figure 8. Elaboration on some lines are provided as follows:

- Function `FINDCONTIGUOUSVECTORS()`: constructs the contiguous vectors recursively by crawling over all atoms that are separated from at least one other atom in the set by a distance no more than $r_{\text{gap}}$. When an atom that is translationally equivalent to one that has previously been visited is
found, then the vector between these atoms is a candidate vector. It is added to the set of contiguous vectors if it is not spanned by the ones already in the set.

Algorithm 4 Detection of periodic sublattice in structures

Input:
- \( \mathbf{x}[] \), \( \{ \mathbf{v}_i \} \) - As defined above.
- seen[] - Array of boolean type. seen[i] indicates whether the atom \( \mathbf{x}[i] \) has been passed to the function FindContiguousVectors() .
- siteIndex - The index of an atom for which neighboring atoms are searched.
- precision - This value is used to address numerical error. Distance with an absolute value below precision are treated as if they are zero.

Output:
- \( \{ \mathbf{v}_i \} \) - Contiguous vectors.

1: function GetContiguousVectors(\( \mathbf{x}[] \), seen[], \( \{ \mathbf{v}_i \} \))
2:   for \( i = 1 \) to \( \mathbf{x}.\text{length} \) do
3:     if seen[i] then
4:       continue;
5:     else
6:       seen[i] = true;
7:       \( \{ \mathbf{v}_i \} = \text{FindContiguousVectors}(\mathbf{x}[], \text{seen}[], i, \{ \mathbf{v}_i \}); \)
8:     return \( \{ \mathbf{v}_i \} \);

9: function FindContiguousVectors(\( \mathbf{x}[] \), seen[], siteIndex, \( \{ \mathbf{v}_i \} \))
10:  \( \mathbf{y}[] \) ← array of neighboring atoms of \( \mathbf{x}[\text{siteIndex}] \), within a radius of \( r_{gap} \);
11:  for \( j = 1 \) to \( \mathbf{y}.\text{length} \) do
12:     \( k \) ← index in \( \mathbf{x}[] \) of the translationally equivalent atom of \( \mathbf{y}[j] \);
13:     if not seen[k] then
14:       seen[k] = true;
15:       \( \{ \mathbf{v}_i \} = \text{FindContiguousVectors}(\mathbf{x}[], \text{seen}[], k, \{ \mathbf{v}_i \}); \)
16:   else
17:     \( \mathbf{u} = \mathbf{y}[j] - \mathbf{x}[k] \);
18:     if \( \| \mathbf{u} \| < \text{precision} \) then
19:       continue;
20:     else if not parallelToLatticePlane(\( \mathbf{u}, \{ \mathbf{v}_i \} \)) then
21:       add \( \mathbf{u} \) to \( \{ \mathbf{v}_i \} \);
22:     if \( \mathbf{v}.\text{length} == 3 \) then
23:       return \( \{ \mathbf{v}_i \} \);
24: return \( \{ \mathbf{v}_i \} \);

Figure 8. Algorithm for detection of the periodic sublattice in structures without periodicity in three dimensions.

- GetContiguousVectors() ensures that all atoms in the unit cell are visited by FindContiguousVectors(). This is important for situations in which there are multiple sets of contiguous atoms separated by at least \( r_{gap} \) (e.g. a molecule above a slab, or something like Figure 7b).
Once we have identified the contiguous vectors, the vacuum direction(s) are calculated as the directions that are normal to all contiguous vectors. The algorithm then distorts the input structure by stretching the primitive lattice vectors along the vacuum directions so that their projections along the vacuum directions have sizes at least \( r_{\text{min}} \) (2\( r_{\text{min}} \) for nanowires and slabs). The components of the lattice vectors parallel to the contiguous vectors are not changed. This effectively tells the lattice-generation algorithm that spacing between translationally equivalent atoms is already sufficiently large in the vacuum directions, and supercells only need to be created in the directions parallel to the contiguous vectors. The grid-generation algorithm is used on the distorted structure. The coordinates of the generated \( k \)-points, in the basis of reciprocal lattice vectors, are the same for both the original and distorted structure. Through this approach we are able to generate low-dimensional grids that respect the symmetry of the three-dimensional calculation. The complete algorithm is summarized as pseudocode in Figure 9. Explanations for some lines are presented as follows:

- Line 3: \( V_{\text{min}} \) represents the minimum supercell volume. It’s equal to the volume of a primitive unit cell, \( V_p \), times the minimum number of total \( k \)-points, \( N_{\text{min}} \), that users specify.
- Line 4 – line 11: \( r_{\text{stretch}} \) is the target distance by which the projections of primitive lattice vectors along the vacuum directions should be stretched to. This block demonstrates how to calculate this value for various periodicities. For \( n = 0 \), \( r_{\text{stretch}} \) is equal to the larger value between \( r_{\text{min}} \) and the maximum possible value of \( r_{\text{lattice}} \) for a unit cell volume of \( V_{\text{min}} \). The latter is achieved when the lattice is close-packed with a fcc structure. For \( n = 1 \), line 7 gives the minimum length of the real-space superlattice vectors parallel to the one-dimensional structure. Line 13 then calculates \( r_{\text{stretch}} \) by finding the larger value between \( r_{\text{min}} \) and the maximum possible value of \( r_{\text{lattice}} \) for a two-dimensional lattice with primitive cell area of \( V_{\text{min}} / r_{\text{periodic}} \). The latter is achieved for a hexagonal lattice. For \( n = 2 \), line 10 calculates the maximum of 1) the minimum cell area for a planar lattice for which \( r_{\text{lattice}} \) is at least \( r_{\text{min}} \) and 2) the cell area for the lattice formed by the found contiguous vectors. The area given by 1) can be calculated by assuming the 2-dimensional lattice is hexagonal.
- Line 13 – line 15: this code block calculates a uniform scaling ratio for all lattice vectors. The maximum ratio is selected to ensure the projections of all lattice vectors along vacuum directions have a length at least \( r_{\text{stretch}} \).
Algorithm 5 Adjusting structures without three-dimensional periodicity

Input:
- \( \mathbf{x} \) - Array of atomic coordinates in the basis of \( \{ \mathbf{a}_1', \mathbf{a}_2', \mathbf{a}_3' \} \).
- \( r_{\text{gap}} \) - Minimum gap size between atoms, larger than which a gap is considered as a vacuum.
- \( N_{\text{min}} \) - The minimum number of total k-points specified by users.
- \( \{ \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3 \} \) - Primitive lattice vectors.
- \( V_p \) - Volume of the primitive unit cell.

Output:
- \( \{ \mathbf{a}_1', \mathbf{a}_2', \mathbf{a}_3' \} \) - Stretched lattice vectors whose projections along the vacuum directions having sizes at least the \( 2 \times r_{\text{min}} \).

Initialization:
- \( \{ \mathbf{v}_i \} \) - An empty array of contiguous vectors.
- \( \text{seen}[\cdot] \) - An array of boolean type with \( x \).length elements initialized as false.

1: \( \{ \mathbf{v}_i \} = \text{getContiguousVectors}(\mathbf{x}[\cdot], \text{seen}[\cdot], \{ \mathbf{v}_i \}) \);
2: \( n \leftarrow \) dimensions of \( \{ \mathbf{v}_i \} \);
3: \( V_{\text{min}} = V_p \times N_{\text{min}} \);
4: if \( n == 0 \) then
5: \( r_{\text{stretch}} = \max(r_{\text{min}}, \sqrt[3] {V_{\text{min}}/\sqrt{2}}) \);
6: else if \( n == 1 \) then
7: \( r_{\text{periodic}} = \max(r_{\text{min}}, ||\mathbf{v}_i||) \);
8: \( r_{\text{stretch}} = 2 \times \max(r_{\text{min}}, \sqrt{(2\sqrt{3})/3 \cdot V_{\text{min}}/r_{\text{periodic}}}) \);
9: else if \( n == 2 \) then
10: \( S_{\text{periodic}} = \max(\sqrt{3}/2 \cdot r_{\text{min}}, ||\mathbf{v}_1 \times \mathbf{v}_2||) \);
11: \( r_{\text{stretch}} = 2 \times \max(r_{\text{min}}, V_{\text{min}}/S_{\text{periodic}}) \);
12: scalingRatio = 0;
13: for \( \mathbf{a}_i \) in \( \{ \mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3 \} \) do
14: \( \mathbf{n}_i \leftarrow \) projection of \( \mathbf{a}_i \) to the normal directions of lattice \( \{ \mathbf{v}_i \} \);
15: scalingRatio = \max(scalingRatio, \frac{r_{\text{stretch}}}{||\mathbf{n}_i||})
16: \( \{ \mathbf{e}_i \} \leftarrow \) unit vectors normal to \( \{ \mathbf{v}_i \} \);
17: for \( \mathbf{a}_i \) in \( \{ \mathbf{a}_i \} \) do
18: for \( \mathbf{e}_j \) in \( \{ \mathbf{e}_i \} \) do
19: \( \mathbf{a}_i' = a_i + [(\text{scalingRatio} - 1) \cdot (a_i \cdot e_j)] \cdot e_j \);
20: return \( \{ \mathbf{a}_1', \mathbf{a}_2', \mathbf{a}_3' \} \);

Figure 9. Algorithm for stretching lattice vectors to reduce k-point density along the vacuum directions accordingly.

2.7 Storing and retrieving k-point grids from the database

Section 2.1 to section 2.5 describe algorithms for dynamical generation of the optimal k-point grid. To further accelerate the search of the optimal grid, we have employed the above algorithms to generate a database of optimal grids. The use of this database can significantly speed up the generation of a generalized k-point grid. An overview of the database has been presented in our previous work [4]. Since publishing that work, the number of possible shift vectors for grids contained in the database has been
increased, and the maximum size of stored \( k \)-point grids has increased from \( 1,728 \times 12 \times 12 \times 12 \) to \( 5,832 \times 18 \times 18 \times 18 \) for orthorhombic, tetragonal, trigonal and hexagonal systems. The maximum size for cubic systems has grown to \( 46,656 \times 36 \times 36 \times 36 \).

The database currently contains 428,632 pre-calculated symmetry-preserving grids, both shifted and \( \Gamma \)-centered, for each of the 21 centrosymmetric symmorphic space groups other than triclinic and monoclinic ones. The number of grids has increased by 63.7\%, compared with the database used in our previous work [4] as we have increased grid density and the number of shift vectors considered. The grids for each of these space groups are stored in 42 separate binary files (21 for shifted and 21 for \( \Gamma \)-centered grids). Figure 10 gives a schematic outline of the database organization. We present the algorithms for mapping \( \mathbf{H} \) to a unique index, estimating \( r_{\text{lattice}} \) from sets of coefficients, and searching the database below.

Figure 10. Schematic representation of the database organization. “m\_IncludeGamma” specifies whether the grids in this file contain the \( \Gamma \) point. “m\_MaxKnownGridSize” indicates the maximum size of the grids. “num\_Lattice” is the total number of \( k \)-point grids stored in this file. “m\_Lattices” is an array of lists of grids. Grids with the same \( N_i \) are stashed in the same list, and the lists are ordered by \( N_i \). Each “KnownLattice” entry in the list represents a \( k \)-point grid. “m\_Size” and “m\_NumDistinctKPoints” represent the total number of \( k \)-points (\( N_F \)) and the number of symmetrically irreducible \( k \)-points (\( N_i \)). “m\_LatticeIndex” is a unique index assigned to each superlattice for regenerating the transformation matrix in HNF, \( \mathbf{H} \) (described in more detail below). “m\_ShiftArray” stores the shift vector for the grid in coordinates of the reciprocal lattice for the conventional primitive cell. “m\_KnownCoefficients” is an array of coefficients for quickly estimating the potential value of \( r_{\text{lattice}} \).
2.7.1 Assigning superlattice indices to $H$ and regenerating $H$ from an index

As shown by equation (2), each $k$-point grid uniquely corresponds to a real-space superlattice. Section 2.2 shows that each superlattice can be uniquely represented by a transformation matrix $H$, which is $M$ in Hermite normal form. For a given lattice size, all possible matrices in Hermite normal form can be systematically generated by enumerating all possible factor sets and, for each factor set, iterating over all possible values of the off-diagonal elements. This presents a straightforward algorithm for assigning a unique index to any matrix in Hermite normal form for which the lattice size (the determinant of the matrix) and the dimensionality (the number of rows and columns in the matrix) are known.

We first illustrate our approach via an example. Suppose we would like to generate an index for a three-dimensional superlattice of with 15 primitive cells per supercell. We start by systematically listing all the possible permutations of ways in which 15 can be factored into three integers:

$$\{1,1,15\}, \{1,3,5\}, \{1,5,3\}$$
$$\{1,15,1\}, \{3,1,5\}, \{3,5,1\}$$
$$\{5,1,3\}, \{5,3,1\}, \{15,1,1\}$$

Each set of factors $\{f_1,f_2,f_3\}$ corresponds to the diagonal elements $\{H_{11},H_{22},H_{33}\}$ for the matrix in Hermite normal form. The total number of unique matrices in Hermite normal form for each set of factors is therefore given by the total number of possible combinations for the off-diagonal elements.

$$\text{number of matrices per factor set} = H_{11}^2H_{22} = f_1^2f_2$$

(16)

For a matrix constructed from a given factor set, we can assign a unique index from 0 to $H_{11}^2H_{22} - 1$ based on the values of the off-diagonal elements using

$$\text{index within factor set} = H_{21} + H_{31}H_{11} + H_{32}H_{11}^2.$$  

(17)

The final index for the matrix is therefore

$$\text{index} = H_{21} + H_{31}H_{11} + H_{32}H_{11}^2 + \sum_{\text{Previous factor sets}} f_1^2f_2.$$  

(18)

As an example, consider the following matrix in Hermite normal form, created using the 8-th factor set from those listed in equation (15).
the factorizations precede the set of its diagonal elements are

\[
\left\{\{1,1,15\},\{1,3,5\},\{1,5,3\},\{1,15,1\}\right\}
\]

(20)

The total number of possible HNF matrices with diagonal elements being one of these factorizations can be calculated by

\[
\text{number of matrices} = \sum_{\text{Previous factor sets}} f_1^2 f_2
\]

\[
=1^2 \times 1 + 1^2 \times 3 + 1^2 \times 5 + 1^2 \times 15 + 3^2 \times 1 + 3^2 \times 5 + 5^2 \times 1
\]

\[
= 1 + 3 + 5 + 15 + 9 + 45 + 25
\]

\[
= 103
\]

The number of matrices (inclusive) precede the given \( H \) can be calculated by

\[
\text{index within factor set} = H_{21} + H_{31}H_{11} + H_{32}H_{11}^2
\]

\[
= 4 + 0 \times 5 + 2 \times 5^2
\]

\[
= 54
\]

Therefore, the index for the given matrix is \( 103 + 54 = 157 \).

Pseudocode for this process is provided in Figure 11. Elaboration on this pseudocode is provided as follows:

- This algorithm is applicable to any dimensions, not restricted to three.
- Line 5: each set of factors composes a possible set of diagonal elements of HNF matrices with determinant \( N_T \). Permutations of factors are counted as different sets, since they reside on different diagonal positions of \( H \). The array of factor sets is arranged in ascending order in terms of the value of each factor. For example, for \( N_T = 15 \), \( \{1,1,15\} \) is the first factorization. \( \{1,15,1\} \) precedes \( \{3,1,5\} \), since the first factor of the former factorization is smaller than that of the latter.
- Line 7 – line 10: this block counts the cumulative number of matrices that can be generated by the factor sets preceding the factor sets used to construct \( H \) (equation (16)).
• Line 11: the second item of the right-hand side of the equation calculates the index of a matrix within a given factor set (equation (17)).

Algorithm 6 Assigning superlattice indices based on transformation matrices in HNF

| Input: |
|--------|
| H - Transformation matrix in Hermite normal form (HNF). |
| n - Dimension of the matrix. |

| Output: |
|--------|
| index - Index of the superlattice. |

1: if \( n == 1 \) then
2: \( \text{return } H_{11}; \)
3: else
4: \( N_t = \text{det}(H); \)
5: factorSets[][][] \( \leftarrow \) Sets of factorizations of \( N_t \) into \( n \) integral numbers;
6: index = 0
7: for diagonals[] in factorSets[][][] do
8: if diagonals[]= \( \{H_{11}, H_{22}, \ldots, H_{nn}\} \) then
9: \( \text{break;} \)
10: index = index + \( \prod_{i=1}^{n-1} \) diagonals[i] \( ^{n-i} \);
11: index = index + \( \sum_{i=2}^{n-1} \sum_{j=1}^{i-1} [H_{ij} \cdot (\prod_{k=1}^{i-2} H_{kk}^{i-k-1}) \cdot (\prod_{k=1}^{j-1} H_{kk})]; \)
12: \( \text{return } \) index;

Figure 11. Algorithm for assigning superlattice indices based on the HNF of transformation matrices. It is applicable to matrices with any dimensions.

The opposite operation, which returns a transformation matrix in HNF, can be easily derived based on the same indexing logic. The pseudocode of this opposite operation is shown in Figure 12. Take the index calculated above as an example. The input index is 157 and determinant is 15. The index is within the index interval of the cumulative number of matrices for the factorization \( \{5,3,1\} \). The rank of the matrix within this factorization is \( 157 - 103 = 54 \). Then the off-diagonal elements can be calculated by

\[
H_{32} = 54 / (H_{11} \cdot H_{11}) = 54 / 25 = 2 \\
H_{31} = 4 / (H_{11}) = 4 / 5 = 0 \\
H_{21} = 4 / 1 = 4 \\
\]

(23)

The divisions are integer division and the remainder of each division is taken to calculate next off-diagonal element.


Algorithm 7 Regenerating the transformation matrix from an index

Input:
- \( n \) - Dimension of the matrix.
- \( N_T \) - Number of total \( k \)-points.
- \( \text{index} \) - Index of the superlattice.

Output:
- \( \texttt{H} \) - Transformation matrix in Hermite normal form (HNF).

Initialization:
- \( \texttt{H} \) - \( 3 \times 3 \) zero matrix.

1: \( \text{factorSets}[][] \leftarrow \) Sets of factorizations of \( N_T \) into \( n \) integral numbers.
2: \textbf{for} diagonals[] in factorSets[][] \textbf{do}
3:     count = \( \prod_{i=1}^{n-1} \text{diagonals}[i]^{n-1} \);
4:     \textbf{if} index > count then
5:         index = index - count;
6:     \textbf{else}
7:         \textbf{for} \( i = 1 \) to \( n \) \textbf{do}
8:             \( H_{ii} = \text{diagonals}[i] \); \( \triangleright \text{assign diagonal elements} \)
9:         \textbf{break};
10:     \textbf{for} \( i = n \) to 2 \textbf{do}
11:         \textbf{for} \( j = i - 1 \) to 1 \textbf{do}
12:             \( c = \prod_{k=1}^{i-2} (H_{kk}^{j-k-1}) \cdot \prod_{k=1}^{j-1} (H_{kk}) \)
13:             \( H_{ij} = \text{index} / c; \) \( \triangleright \text{integer division} \)
14:             index = index \( \% c; \) \( \triangleright \text{modulo operation} \)
15:     \textbf{return} \texttt{H};

Figure 12. Algorithm for retrieving the superlattice from a given index.

2.7.2 Determination of coefficients and estimation of \( r_{\text{lattice}} \)

Ensuring that \( r_{\text{lattice}} \geq r_{\text{min}} \) requires a calculation of \( r_{\text{lattice}} \) for each candidate superlattice. Therefore, determination of \( r_{\text{lattice}} \) by Minkowski reduction could become a major overhead. The database-query approach cuts down the computational cost by remembering which linear combination of primitive lattice vectors resulted in the shortest lattice vector every time it performs a Minkowski reduction on a candidate superlattice. The next time the same generating matrix, \( \texttt{H} \), is encountered, the database first tries the known linear combinations of primitive vectors to see if any of them has a length less than \( r_{\text{min}} \). If they do, the lattice can be eliminated from consideration without performing Minkowski reduction. If not, then full Minkowski reduction is performed. If a new linear combination of primitive vectors is found that has a length less than \( r_{\text{min}} \), the coefficients of this combination are stored for future screens. In this way, the database continuously learns how to improve its performance.
The database remembers the linear combinations of primitive lattice vectors that result in a superlattice vector by projecting the superlattice vector onto a set of pre-defined mutually orthogonal vectors. For cubic, tetragonal, and orthorhombic systems, the orthogonal vectors are the conventional vectors defined in section 2.3.1. For hexagonal and trigonal systems, the orthogonal vectors can be calculated by

$$v_i = c_1, v_3 = c_3$$

$$v_2 = \frac{c_1 \times c_3}{\|c_1 \times c_3\|} \|c_i\| \quad (24)$$

where \(\{c_1, c_2, c_3\}\) are conventional lattice vectors as defined in section 2.3.1, and \(\{v_1, v_2, v_3\}\) are the orthogonal vectors. Given a shortest lattice vector in a superlattice, \(r\), the \(i^{th}\) coefficient is calculated by

$$c_i = \frac{r \cdot v_i}{v_i \cdot v_i} \quad (25)$$

where \(v_i\) is the \(i^{th}\) orthogonal vector. The opposite operation, calculating the length of a stored candidate vector in a superlattice, is accomplished by

$$\|r\| = \sqrt{\sum_{i=1}^{3} (c_i \|v_i\|)^2} \quad (26)$$

As the orthogonal vectors \(\{v_1, v_2, v_3\}\) need only be calculated once for any new query and the all sets of coefficients \(\{c_1, c_2, c_3\}\) are stored in the database, equation (26) provides a rapid way to calculate an upper bound on \(r_{\text{lattice}}\).

### 2.7.3 Grid generation by searching the database

Dynamic grid generation can be slow for requests with large \(r_{\text{min}}\) and \(N_{\text{min}}\). A pre-generated database accelerates grid generation by skipping the non-symmetry-preserving superlattices and the low quality superlattices (e.g. the ones with too many symmetrically irreducible \(k\)-points). Figure 13 provides the workflow of the database-searching approach for generating the optimal generalised \(k\)-point grid. It starts by estimating the lower bound of the number of symmetrically distinct \(k\)-points:

$$N_{\text{lower}} = m \text{MinDistinctKPoints} \left[ \max \left( \frac{\sqrt{2}}{2}, \frac{p_{\text{min}}^3}{V_p}, N_{\text{min}} \right) \right] \quad (27)$$
$m_{\text{MinDistinctKPoints}}[]$ is an integral array created when loading a file from the database. The $N_T$-th element represents the minimum value of $N_i$ of all the grids stored in this file that have a size of $N_T$. The minimum $N_T$ that satisfies $r_{\text{lattice}} \geq r_{\text{min}}$ and $N_T \geq N_{\text{min}}$ is calculated by the $\max()$ function. The first argument is the minimum size of a superlattice that could satisfy $r_{\text{lattice}} \geq r_{\text{min}}$, and the justification of the prefactor is similar to that of equation (8). For $\Gamma$-centered grids of all lattices and shifted grids of non-cubic lattices, the minimum volume is that of a fcc unit cell a distance of nor more than $r_{\text{min}}$ between lattice points. For cubic systems, a fcc superlattice results in a bcc reciprocal lattice, and the only symmetry-preserving shift of the $\Gamma$ point in a bcc lattices is $[1/2, 1/2, 1/2]$, which results in a $\Gamma$-centered grid. Therefore, the minimum volume of supercell for a shifted grid in a cubic system is that of the second closest-packed lattice, a body-centered-cubic (bcc) lattice [30]. $N_{\text{lower}}$ in this case is calculated by

$$N_{\text{lower}} = m_{\text{MinDistinctKPoints}} \left[ \max \left( \frac{4}{3\sqrt{3}} \frac{r_{\text{min}}^3}{V_p} , N_{\text{min}} \right) \right]. \quad (28)$$

$N_{\text{upper}}$ is the maximum size of superlattices allowed when generating the database.

Grids are searched by starting with the list for which $N_i = N_{\text{lower}}$ and incrementally increasing the size of $N_i$ until $N_i = N_{\text{upper}}$. Once an optimal lattice for a given value of $N_i$ is found, there is no need to search for larger values of $N_i$. If the search cannot find a fulfilling grid before $N$ exceeds $N_{\text{upper}}$, it is most likely the required superlattice has a size larger than the maximum grid size that the database currently can provide. An exception is thrown to inform users about the overflow.
Figure 13. A diagram summarizes the grid generation workflow by the database-searching approach.

3. Implementations

Three implementations are provided to meet the diverse demands of users.

3.1 K-Point Grid Server: A ready-to-use online application

The K-Point Grid Server, referred as “the server” below, is a ready-to-use web-based application. It generates the optimal generalized Monkhorst-Pack grids by rapidly searching a pre-generated database, as discussed in section 2.7.3. The database contains GMP grids calculated from all symmetry-preserving superlattices of a set of 16808 sample structures in cubic, hexagonal, trigonal, tetragonal, and orthorhombic crystal systems with different lattice parameters. The maximum size of the superlattices is 5832 (18×18×18) for structures in all lattice systems except for triclinic and monoclinic ones. The ratio
of the longest conventional lattice vector to the shortest one is up to 64. Such dense sampling of the possible lattice parameters should make the database comprehensive enough to cover nearly all input structures from users. The database searching approach saves the computational cost of enumerating the superlattices and counting the symmetrically distinct $k$-points in corresponding grids for every user request, which gives it an advantage over dynamic grid generation. For the monoclinic and triclinic systems, however, the server uses the dynamic searching scheme, since the database searching approach wouldn’t be as beneficial as it is to the other five Bravais lattices because of the huge number symmetry-preserving superlattices there would be for these two systems. The algorithm for detecting vacuum spaces in structure is also implemented in the server to reduce $k$-point accordingly for structures with inherent anisotropy, as are other algorithms for identifying symmetry that are specific to the ab-initio software package being used.

Users can tailor their requests to the server through a set of parameters defined in a PRECALC file, and the server is queried using a small script called “getKPoints”. An example of a PRECALC file, the getKPoints script, and a detailed description of all allowed parameters in PRECALC can be found on our website (http://muellergroup.jhu.edu/K-Points.html).

### 3.2 K-Point Grid Generator: An open-source, stand-alone application

The K-Point Grid Generator is a self-contained application for users with runtime environments that might not have an internet connection. It has the exact same set of functionalities as the server and is updated accordingly every time a new version the server is released. Users still specify parameters through a PRECALC file and launch the application through a script getKPoints. But the script is different from the one used for server and is tailored for the stand-alone application. The Java programming language is used to ensure the portability and a consistent performance across operating systems. The project is open sourced through a public repository (https://gitlab.com/muellergroup/k-pointGridGenerator). A pre-built binary can be downloaded from our website, and is packaged with the tailor calling script getKPoints and with a complete set of files for the database. The database files are stored in binary gzipped format and take up about 7.15 MB of disk space.
3.3 KpLib: A Lightweight, Open-source C++ Library

KpLib is a lightweight library, which implements the dynamic generation approach described in sections 2.1 to 2.5. The C++ programming language is chosen to make the interfacing with other programming languages easier, and thus help integrate with computational materials software packages. The source code only contains 1117 lines, and the API uses elementary data structures as argument types, which should be available in most programming languages and easy to write wrapping functions for. We have written a demonstration application, integrated with spglib [29], to show how to work with the API. The library is also open sourced and a documentation of the API is provided on the homepage of its public repository (https://gitlab.com/muellergroup/kplib).

4. Benchmarking Results

4.1 Acceleration with \( r_{\text{min}} \) being the limiting factor

The algorithms for enumerating symmetry-preserving superlattices in section 2.3 are accelerated by enforcing \( r_{\text{lattice}} > r_{\text{min}} \) at each step of constructing a superlattice. This allows the algorithm to skip many superlattices at an early stage. To measure the degree of acceleration by utilizing the screening of \( r_{\text{min}} \), benchmarking of the computation time for generating a GMP grid was performed for both \( \Gamma \)-centered and shifted grids, and with both the stand-alone Java application and the demonstration application in C++ using kpLib. In each case, GMP grids were generated at three values of \( r_{\text{min}} \) for the benchmark set of 102 structures randomly selected from the Inorganic Crystal Structure Database (ICSD) used in our previous work [4]. Each calculation was repeated five times and the average response time was recorded as the calculation time for that structure. The computation time for each crystal system is taken as the average time of structures within the 102 materials that belongs to this system. The ratio of the computation time between non-accelerated and accelerated codes were computed and plotted in Figure 14. The benchmark was performed on a virtual machine with a Ubuntu 18.04 operation system, and on Intel Core i7-8550U processors with a clock speed of 1.80GHz. Results for both the Java and C++ codes demonstrate a significant acceleration, and the amount of acceleration increases as \( r_{\text{min}} \) grows. Consistent acceleration is observed for both the \( \Gamma \)-centered and shifted grids. The highest ratio for the Java code is 10.6 for \( \Gamma \)-
centered grids in the hexagonal system with $r_{\text{min}}$ equal to 75 Angstroms, and that for the C++ code is 37.4 with $r_{\text{min}}$ equal to 75 Angstroms for $\Gamma$-centered grids in trigonal system.

![Figure 14. Ratios of computation time between non-accelerated and accelerated algorithm for all seven crystal systems with $r_{\text{min}}$ at 25, 50, and 75 Angstroms. The top two charts illustrate results of the stand-alone application in Java, for a) $\Gamma$-centered grids and b) shifted grids. The bottom graphs demonstrate the results of the C++ application using kpLib, for c) $\Gamma$-centered grids and d) shifted grids. The $x$-axis lists the crystal systems. From left to right, they represent triclinic, monoclinic, orthorhombic, tetragonal, trigonal, hexagonal and cubic, respectively.](image)

5. Conclusion

Generalized Monkhorst-Pack $k$-point grids have been shown to be more efficient than traditional Monkhorst-Pack grids [4, 7]. In this work, we presented core algorithms for generating optimal generalized Monkhorst-Pack grids both dynamically and through searching a pre-generated database. The rapidness of our algorithms relies on the recognition that any lattices can be decomposed into stacks of two-dimensional lattices along the primary direction of a structure, and by enforcing the constraint that $r_{\text{lat}} \geq r_{\text{min}}$ while generating symmetry-preserving lattices. Our dynamic generation scheme, which can take any combination of lower bounds for $r_{\text{lat}}$ and $N_T$ as inputs, guarantees that the identified grid is
optimal, in the sense that there are no grids with a smaller number of irreducible \( k \)-points that satisfy the required constraints. We also presented an algorithm to detect and adjust structures that don’t have three-dimensional periodicity. These algorithms are used to populate a database for even faster generation of the optimal GMP grids. We demonstrate a way to assign a unique index to each superlattice, allowing for fast regeneration of superlattices without storing each transformation matrix. The database search is accelerated by quickly estimating \( r_{\text{lattice}} \) from stored coefficients, which represent the shortest lattice vectors in superlattice defined by the same \( \mathbf{H} \) and appeared in past grid generations. Various implementations, in the forms of a web-based server, a stand-alone application, and a light-weight library, are provided to fulfill the demands of users in various computing environments. Both the \( K \)-Point Server and the \( K \)-point Grid Generator implement all the algorithms presented in this work, using the database-query approach for lattices other than the triclinic and monoclinic ones. For the excluded two crystal systems, the dynamic generation approach is adopted. The \( C++ \) library, kpLib, implements the dynamic generation scheme and is provided for integration with \textit{ab initio} simulation packages. The total number of lines of source code in kpLib is only 1117 so that it should be relatively easy to translate into other programming languages and implementations. The \( K \)-point Grid Generator and kpLib are open-sourced through online repositories.
Acknowledgements

The authors thank the National Science Foundation for the financial support under the Award No. DMR-1352373, the Maryland Advanced Research Computing Center (MARCC) for providing computational resources, and Gus Hart for helpful discussions.

References

1. Monkhorst, H.J. and J.D. Pack, Special Points for Brillouin-Zone Integrations. Physical Review B, 1976. 13(12): p. 5188-5192.
2. Ashcroft, N.W. and N.D. Mermin, Solid State Physics. 1976, Belmont, USA: Brooks/Cole. 132-143.
3. Tinkham, M., Group Theory and Quantum Mechanics. 1964, New York: McGraw-Hill Book Company. 279-281.
4. Wisesa, P., K.A. McGill, and T. Mueller, Efficient generation of generalized Monkhorst-Pack grids through the use of informatics. Physical Review B, 2016. 93(15).
5. Froyen, S., Brillouin-zone integration by Fourier quadrature: Special points for superlattice and supercell calculations. Phys Rev B Condens Matter, 1989. 39(5): p. 3168-3172.
6. Choudhary, K. and F. Tavazza, Convergence and machine learning predictions of Monkhorst-Pack k-points and plane-wave cut-off in high-throughput DFT calculations. Computational Materials Science, 2019. 161: p. 300-308.
7. Morgan, W.S., et al., Efficiency of Generalized Regular k-point grids. Computational Materials Science, 2018. 153: p. 424-430.
8. Cao, L., et al., Mechanistic insights for low-overpotential electroreduction of CO2 to CO on copper nanowires. ACS Catalysis, 2017. 7(12): p. 8578-8587.
9. Ding, Y. and Y. Wang, Tunable Electronic Structures of Hydrogenated Zigzag and Armchair Dumbbell Silicene Nanosheets: A Computational Study. The Journal of Physical Chemistry C, 2018. 122(40): p. 23208-23216.
10. Greenman, K.P., L. Williams, and E. Kioupakis, Lattice-Constant and Band-Gap Tuning in Wurtzite and Zincblende BnGaN Alloys. arXiv preprint arXiv:1905.00467, 2019.
11. Hernandez, A., et al., Fast, accurate, and transferable many-body interatomic potentials by genetic programming. arXiv preprint arXiv:1904.01095, 2019.
12. Kratzer, P. and J. Neugebauer, The basics of electronic structure theory for periodic systems. Frontiers in chemistry, 2019. 7.
13. Li, C., et al., Improved Prediction of Nanoalloy Structures by the Explicit Inclusion of Adsorbates in Cluster Expansions. The Journal of Physical Chemistry C, 2018. 122(31): p. 18040-18047.
14. Liu, P., et al., Active basal plane catalytic activity and conductivity in Zn doped MoS2 nanosheets for efficient hydrogen evolution. Electrochimica Acta, 2018. 260: p. 24-30.
15. Liu, Y., et al., Synthesis and Anisotropic Electrocatalytic Activity of Covellite Nanoplatelets with Fixed Thickness and Tunable Diameter. ACS applied materials & interfaces, 2018. 10(49): p. 42417-42426.
16. Lu, X., et al., Bandgap control and optical properties of β-Si3N4 by single-and co-doping from a first-principles simulation. International Journal of Modern Physics B, 2018. 32(14): p. 1850178.
17. Nyshadham, C., et al., Machine-learned multi-system surrogate models for materials prediction. npj Computational Materials, 2019. 5(1): p. 51.
18. Oses, C., et al., AFLOW-CHULL: cloud-oriented platform for autonomous phase stability analysis. Journal of chemical information and modeling, 2018. 58(12): p. 2477-2490.

19. Raciti, D., et al., Low-overpotential electroreduction of carbon monoxide using copper nanowires. ACS Catalysis, 2017. 7(7): p. 4467-4472.

20. Rosenbrock, C.W., et al., Machine-learned Interatomic Potentials for Alloys and Alloy Phase Diagrams. arXiv preprint arXiv:1906.07816, 2019.

21. Tantardini, C. and A.A. Michalchuk, Dess - martin periodinane: The reactivity of a λ5 - iodane catalyst explained by topological analysis. International Journal of Quantum Chemistry, 2019. 119(6): p. e25838.

22. Williams, L. and E. Kioupakis, BInGaN alloys nearly lattice-matched to GaN for high-power high-efficiency visible LEDs. Applied Physics Letters, 2017. 111(21): p. 211107.

23. Wolloch, M., D. Suess, and P. Mohn, Influence of antise defects and stacking faults on the magnetocrystalline anisotropy of FePt. Physical Review B, 2017. 96(10): p. 104408.

24. Hart, G.L.W., et al., A robust algorithm for k-point grid generation and symmetry reduction. Journal of Physics Communications, 2019. 3(6): p. 065009.

25. Morgan, W.S., et al., Generalized Regular k-point Grid Generation On The Fly. arXiv preprint arXiv:1902.03257, 2019.

26. Mueller, T.K., Computational studies of hydrogen storage materials and the development of related methods, in Department of Materials Science and Engineering. 2007, Massachusetts Institute of Technology: Boston, Massachusetts. p. 127-132.

27. Hart, G.L.W. and R.W. Forcade, Algorithm for generating derivative structures. Physical Review B, 2008. 77(22).

28. Wisesa, P., K.A. McGill, and T. Mueller, Efficient generation of generalized Monkhorst-Pack grids through the use of informatics. Physical Review B, 2016. 93(15): p. 155109.

29. Togo, A. and I. Tanaka Spglib: a software library for crystal symmetry search. arXiv e-prints, 2018.

30. Giacovazzo, C., et al., Fundamentals of Crystallography. third ed. IUCr Texts on Crystallography. 2002, Oxford: Oxford University Press. 842.

31. Moreno, J. and J.M. Soler, Optimal Meshes for Integrals in Real-Space and Reciprocal-Space Unit Cells. Physical Review B, 1992. 45(24): p. 13891-13898.

32. Nguyen, P.Q. and D. Stehlé, Low-Dimensional Lattice Basis Reduction Revisited. 2004. Berlin, Heidelberg: Springer Berlin Heidelberg.